
AFRL-RI-RS-TR-2014-142

# MULTI-CORE PROGRAMMING DESIGN PATTERNS: STREAM PROCESSING ALGORITHMS FOR DYNAMIC SCENE PERCEPTIONS

UNIVERSITY OF MISSOURI

*MAY 2014*

FINAL TECHNICAL REPORT

***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED***

STINFO COPY

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-142 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

**/ S /**
STANLEY LIS
Work Unit Manager

**/ S /**
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| MAY 2014 | FINAL TECHNICAL REPORT | NOV 2010 – NOV 2013 |

**4. TITLE AND SUBTITLE**

MULTI-CORE PROGRAMMING DESIGN PATTERNS: STREAM PROCESSING ALGORITHMS FOR DYNAMIC SCENE PERCEPTION

**5a. CONTRACT NUMBER**
FA8750-11-1-0073

**5b. GRANT NUMBER**
N/A

**5c. PROGRAM ELEMENT NUMBER**
62788F

**6. AUTHOR(S)**

Kannappan Palaniappan

**5d. PROJECT NUMBER**
T2MC

**5e. TASK NUMBER**
MI

**5f. WORK UNIT NUMBER**
SS

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Missouri
316 University Hall
Columbia, MO 65211-3020

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Air Force Research Laboratory/RITB
525 Brooks Road
Rome NY 13441-4505

**10. SPONSOR/MONITOR'S ACRONYM(S)**
AFRL/RI

**11. SPONSOR/MONITOR'S REPORT NUMBER**
AFRL-RI-RS-TR-2014-142

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

We have implemented, tested, validated and benchmarked a scalable parallel implementations of the integral histogram algorithm critical for computer vision tasks for fast multiscale subwindow-based object searching, motion analysis and content-based image retrieval applications. Several integral histogram kernels using CUDA optimizations for many core GPUs were investigated. The integral histogram algorithm was also parallelized using the StarSs programming model in collaboration the Barcelona Supercomputing Center for several architectures including Cell/B.E., GPU and SMP. The Cell/B.E. implementation of the integral histogram using cross-weave scan and 16 bins for a 640x480 image reaches 160 fr/sec using 8 SPEs. The wavefront scan for the same sized image reaches almost 200 fr/sec but critically depends on the block size. The GPU implementation of the integral histogram was 60 times faster than the sequential CPU version for a 1K x 1K image reaching 49 fr/sec and 21 times faster for 512 x 512 images reaching 194 fr/sec. The implemented code has been delivered to AFRL for transition to other programs like CETE.

**15. SUBJECT TERMS**
parallel multicore algorithms, integral histogram, computer vision, graphics processing unit (GPU), IBM Cell Broadband Engine, StarSs

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | **STANLEY LIS** |
| U | U | U | UU | 41 | 19b. TELEPHONE NUMBER (*Include area code*) **N/A** |

## Contents

## List of Figures

## List of Tables

# 1. SUMMARY

The University of Missouri (MU) completed the parallel implementations of several core computer vision algorithms including the integral histogram for fast subwindow object searching that is scalable to large images and large subwindow sizes using Compute Unified Device Architecture (CUDA) for many core Graphics Processing Units (GPU) with various kernel optimizations. The integral histogram algorithm was also parallelized for the Cell Broadband Engine (Cell/B.E.) architecture jointly with the Barcelona Supercomputing Center. These algorithms and code implementations have been delivered to AFRL for transition to other AFRL programs including Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C4ISR) Enterprise to the Edge (CETE), Multi-INT Enhanced Exploitation and Analysis Tools (E2AT) and Next Generation Wide Area Motion Imagery (WAMI). An initial implementation of the 3D spatiotemporal median filter for background model-based fast motion detection, using integral histograms, was also tested.

The rapid succession of powerful and innovative architectures has rekindled an interest in programming models such as CUDA or OpenCL. Parallel programming practice traditionally tends towards thread or streaming models. Threads and streams deliver good performance if the application can be expressed as a set of cooperating parallel resources or restrictions on data access are honored, respectively. We compare this tradition to an approach inspired by the Instruction-Level Parallelism (ILP) in superscalar processors, as found in Star Superscalar[4] (StarSs) and StarPU.[5] These models use a standard, sequential programming language and rely on data dependence analysis to execute the workload in parallel. This results in good portability and simplifies code development. We illustrate these features and describe a parallel implementation of the integral histogram in StarSs. This report provides performance results for the portable implementation of an efficient integral histogram computation in StarSs for symmetric multiprocessing (SMP), the Cell Broadband Engine (Cell/B.E.) and GPU.

We have completed the GPU parallel multicore implementation of the integral histogram which is an extension of the integral image computation for full motion video processing. Several different kernels were implemented that tradeoff compute intensive versus communication intensive approaches. We are characterizing these kernels and are continuing evaluating the performance of the algorithms on different sized images.

The integral histogram for images is an efficient preprocessing method for speeding up diverse computer vision algorithms including object detection, appearance-based tracking, recognition and segmentation. We have completed an efficient GPU implementation based on cross-weave scans implemented using parallel prefix-sums and image transposition. Our proposed Graphics Processing Unit (GPU) implementation uses parallel prefix sums on row and column histograms in a cross-weave scan with high GPU utilization and communication-aware data transfer between CPU and GPU memories.

Parallel implementations of the integral histogram computation were developed, for the multicore Cell/B.E. and many core GPU using CUDA. The Cell/B.E. implementation using cross-weave scan and 16 bins for a 640x480 image reaches 160 fr/sec using 8 Synergistic Processing Elements (SPEs). The wavefront scan for the same parameters reaches almost 200 fr/sec but is more critically dependent on the block size.

Several GPU CUDA kernels for the integral histogram were implemented, tested and evaluated. Two different data structures and communication models were evaluated in these three kernels. A 3-D array to store binned histograms for each pixel and an equivalent linearized 1-D array, each with distinctive data movement patterns. Using the 3-D array with many kernel invocations and low workload per kernel was inefficient, highlighting the necessity for careful mapping of sequential algorithms onto the GPU. We characterized the dependence on image size, tile size, histogram bin size, memory utilization, thread utilization, and data transfer times. We compared the efficiency of the BSC StarSs programming model to extend the integral histogram to SMP and GPU architectures compared to manually optimized GPU CUDA implementations of the integral histogram.

The reorganized 1-D array with a single data transfer to the GPU with high GPU utilization, was 60 times faster than the sequential CPU version for a 1K x 1K image reaching 49 fr/sec and 21 times faster for 512 x 512 images reaching 194 fr/sec. With larger image sizes data transfer communication time dominates a larger percent of the total time taken for the integral histogram task. The integral histogram module is applied as part of the likelihood of features tracking (LOFT) system for video object tracking using fusion of multiple cues.

We have extended the integral histogram approach to design and implement an initial version of the 3D spatiotemporal median filter algorithm for fast motion detection in full motion video. We characterized the performance of the spatiotemporal adaptive x-y-t median operator for moving object detection in complex electrooptical and infrared video sequences. Initial results of the GPU implementation are promising showing performance improvements similar to the integral histogram speedup. We characterized performance of the parallel 3D median filter for different image sizes and varying number of histogram bins and evaluated motion detection performance on real video sequences.[6]

Future work will include integration of these parallel computer vision algorithms with wide area motion imagery and full motion video multitarget tracking systems. We are also working on extending the integral histogram for axis aligned bounding boxes to support oriented bounding boxes.

This report based on several publications written as part of the project deliverables.[6–9] The StarSs programming model is shown to be very efficient for rapid prototyping of portable parallel versions of image processing and computer vision algorithms. However, for best performance manual code tuning and optimization are still necessary at the expense of additional development time and cost.

## 2. INTRODUCTION AND ARCHITECTURE BACKGROUND

### 2.1 Multicore Programming Design Patterns

The Cell Broadband Engine (Cell/B.E.) multicore processor developed by IBM and other companies, incorporates the verb—POWER5— processor as the Power Processor Element (PPE), one of the early general purpose multicore processors, where it cooperates with 8 single instruction multiple data (SIMD) cores or Synergistic Processing Elements (SPEs) to deliver an power efficient single-precision peak performance of more than 256 GFlops. Substantially more raw power became available later, when nVIDIA released the first version of CUDA together with its line of Tesla GPUs. Over the course of the past decade, we have witnessed the rise of homogeneous and heterogeneous multicore processors. These parallel architectures offer affordable and power efficient computing resources for computer vision algorithms in constrained environments. However, programming models for these parallel hardware architectures have not kept pace.

Architectural innovations as these, that affect Thread-Level Parallelism (TLP) or Data-Level Parallelism (DLP), invariably trigger changes in the way we write programs. In the same vein as the development of SMP lead to Open Multi-Processing (OpenMP) and clusters inspired the use of the Message Passing Interface (MPI), the multicore architectures sparked an interest in suitable programming models. Cilk++[10] for example, implements the nested threading model of Cilk[11] for multicores. OoOJava[12] adheres to the same fork-join model and performs automatic dependence analysis during compilation and execution. Its compiler conservatively estimates dependencies in order to build run-time queues of threads. Light-weight checks then enforce a correct schedule of the threads at run-time. CUDA[13] models the computation as a sequence of two-dimensional *thread grids*. Each thread grid consists of three-dimensional, independent *thread blocks*. The threads in a thread block execute the same *kernel* in parallel. Computations and data transfers can be ordered by defining *streams* and the CUDA interface provides explicit control over the memory hierarchy. BrookGPU[14] implements the Brook stream programming language for GPUs. A *stream* identifies a collection of elements and has a shape or dimensionality. A *kernel* applies a function to each element of the stream in parallel. Brook has a reduction construct and is able to infer the size of an output stream based on the kernel's input streams. The OpenCL[15] framework publishes the hardware resources of a heterogeneous system via a uniform interface. Programs written in OpenCL can therefore call kernels that execute in a CPU, GPU or multicore.

These programming models impose an explicit parallel structure on the computation. A thread model exposes individual execution contexts and in that capacity it ties in closely with the underlying hardware. The low level of the thread interface delivers the full computational power of the hardware, but exposes its complexity and quirks as well. Nested thread models further restrict the structure of the computation to the spawn tree of the parallel threads, but not all applications lend themselves to a description in terms of recursive calls to threads.[16] Unless the application exhibits massive amounts of unstructured parallelism one must also provide synchronization between the threads. This requirement stems from the order among the data accesses of the application, whereas the thread model is designed around the (parallel) capacity to process the data. The

former fundamentally characterizes the computation itself, whereas the latter is an artefact of the particular programming model. Parallel computing by definition requires the availability of multiple resources, but in the presence of this cheap commodity the limiting factor is *data dependence*. Streaming models on the contrary make no assumptions about the available resources but they limit the permissible data dependencies. Specifically, kernels process the elements of a stream in parallel, which precludes the existence of data dependencies among the elements.

In the threading model and the stream model the user consequently must understand the data dependencies that govern the computation and take appropriate measures. Either the algorithm must be restructured to avoid synchronization or one must add synchronization primitives, as both models support the fundamental property of data dependence only indirectly. This observation did not in the least prevent the widespread and successful use of these models. The lack of data dependence analysis and enforcement creates the freedom to craft efficient programs tailored for a specific architecture, although the effort involved can be prohibitive. Alternatively the parallel programming model itself can define the computation directly in terms of the data dependencies. StarSs (Section 2.3), StarPU[5] and PLASMA[17,18] are exponents of this trend. Instead of code that explicitly unveils DLP, be it via threads or streams, the aforementioned programming models resort to a *sequential description*, which identifies **tasks** or units of parallel computation. StarSs supports standard C and Fortran, for example. The data dependencies present in the sequential code define **task dependencies**. These are recorded in a **Task Dependency Graph** (TDG) or a similar data structure during execution. The runtime library schedules tasks from the TDG to the parallel resources and enforces the partial order on the tasks. DAGuE[19] is a distributed scheduler for MAGMA[20] that uses a compact and problem-size independent format for the TDG called Job Definition Format (JDF). Like the Parameterized Task Graph,[21] the JDF representation can be used at run time to determine task dependencies without unrolling the TDG of the application and without a centralized arbiter.

Such parallel programming models typically reduce the programming effort, as dependence analysis, scheduling and resource synchronization are handled by the compiler or the runtime instead of by the user. The resulting code also is more readily portable, as the program description (e.g. standard C for a StarSs application) is architecture-independent. This form of organizing a parallel computation bears a striking resemblance to the instruction-level parallelism in superscalar processors. In both cases the workload consists of a sequential stream of tasks or instructions. A superscalar processor tracks the dependencies for a window of instructions and issues instructions without outstanding dependencies. Independent instructions execute in parallel, given the availability of sufficient resources.[22,23]

In this report we describe a parallel implementation of the integral histogram[24] (IH) using the StarSs programming model. It is not our intention to make a case for one programming model or the other. Rather, to evaluate the merits of our implementation, we demonstrate the portability of IH in StarSs by executing the same source code on three different platforms, including SMP, the Cell Broadband Engine (Cell/B.E.) and a GPU platform. Section 4.1 discusses the performance on each architecture. For the Cell/B.E. we compare IH in StarSs with an optimized, hand-coded version (Section 3.3) to get an idea of the relative performance or efficiency of a StarSs application. First we give an overview of StarSs (Section 2.3) and briefly discuss IH (Section 3.1) together with related work on the topic (Section 2.2), followed by the parallel implementation of IH in StarSs (Section 3.2). Section 4.2 formulates some conclusions and hints at future work.

## 2.2 Dynamic Scene Analysis Using Integral Histograms

Histograms limited to an image segment (or *regional histograms*) are widely used in a variety of computer vision tasks. Their application extends from object recognition and image content-based retrieval to segmentation, detection and tracking. Sliding-window search methods can use histogram measures to produce high-quality results but the computational cost is immense. The integral histogram[24] is a recently proposed preprocessing technique that abates said cost. It allows for histogram construction of arbitrary rectangular gridded regions (i.e. images or volumes) in constant time. Preprocessed images are suitable for exhaustive global search using sliding window-based histogram optimization measures, yield high-quality results[25] and still obey real-time bounds. Fast histogram computation using the integral histogram (IH) speeds up the sequential implementation by up to five orders of magnitude. However, the overall cost remains prohibitive for real-time applications with large images, large search window sizes and a large number of histogram bins. For example, a $512 \times 512$ image

search using a 1000-bin feature histogram requires about 1 gigabyte (GB) of memory and takes about one second.[26] The computation of such dense confidence maps remains infeasible for these types of applications. A parallel implementation of the IH would enable methods using global optimization of histogram measures to be competitive with or faster than other approaches in terms of speed. There are a variety of commodity multicore architectures currently available for the parallelization of image- and video-processing algorithms, including IBM's Cell/B.E., GPUs from NVidia and AMD and many-core CPUs from Intel.[27] The vast growth of digital video content has been a driving force behind active research into exploiting heterogeneous multi-core architectures for computationally intensive, multimedia analysis tasks. These include real-time (and super-real-time) object recognition, object tracking in multi-camera sensor networks, stereo vision, information fusion, face recognition, biometrics, image restoration, compression, etc.[28–34]

Data dependence and implicit parallelism are central ideas in StarSs and its runtime controls the scheduling of tasks and data transfer of task arguments. Then the implementations of these aspects in StarSs critically affect the performance of an application. Dynamic dependence analysis can be limited to blocks or more complete based on linear representations.[35] The scheduler is distributed and uses job stealing[16] or speculative techniques[36] for good scalability and to avoid bottlenecks. Dependence tracking and scheduling cooperate to exploit temporal locality.[37] This ability is crucial to circumvent memory bandwidth limitations on multicore- and GPU architectures. In particular, the implementation of StarSs for GPU[38] is able to manage CUDA streams for overlapping communication and computation.

### 2.3 Integral Histogram Using Star Superscalar (StarSs) for Parallel Architectures

The StarSs programming model[4,39,40] provides an environment for the development of portable parallel applications for a variety of architectures. In this report we use the implementations of StarSs for SMP, Cell/B.E. and NVIDIA GPU, referred to respectively as **SMPSs**, **CellSs** and **GPUSs**. The programming model advocates the development of code in a standard, sequential language, such as C or Fortran. On the user's side there are no explicit parallel constructs, like threads or streams. It generally suffices that the user adds pragmas to the original code to mark the code intended to execute on the parallel resources, referred to as **tasks**, typically corresponding to functions or inline code blocks. The StarSs source-to-source compiler converts these pragmas into calls to the StarSs runtime library. As the application advances the StarSs runtime executes the tasks in parallel as dictated by the data dependencies present in the original program (Figure 1).

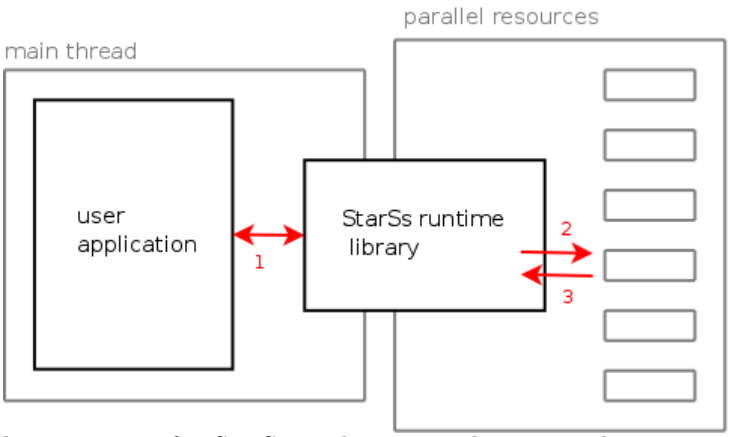



Figure 1: Overview of the structure of a StarSs application. The user code generates tasks according to the sequential program (1), the runtime records the tasks into a TDG, schedules tasks to resources (2) and removes finished tasks from the TDG (3).

In practice the main thread of a StarSs application executes the sequential code and switches to the StarSs library when it encounters code marked with a pragma. The library or runtime does not immediately execute the task. Instead it analyzes the task arguments to find the true dependencies that define task precedence. StarSs avoids output dependencies and anti-dependencies by renaming arguments. The main thread returns control to the user application and the StarSs runtime records the task in the Task Dependence Graph (TDG).

Simultaneously the runtime schedules **ready tasks** (or tasks without outstanding dependencies in the TDG) to the multiple resources. The resources in turn remove finished tasks from the TDG and update the state of dependent tasks. Ultimately this pruning of dependencies updates dependent tasks to ready tasks that again become scheduling candidates.

In StarSs dependence analysis, renaming, scheduling and updates to the TDG take place concurrently during execution. This combination converts a sequential stream of tasks generated by a single thread into a TDG into a parallel execution of tasks on multiple resources. The concept has a strong resemblance to dynamic scheduling in superscalar processors, where the pipeline decodes instructions in order but schedules them to multiple units in parallel. In both cases the ability to track data dependencies drives the parallel execution. The StarSs model hides the underlying hardware and frees the user from the tedious requirements that traditionally accompany parallel programming (synchronization, scheduling, decomposition, load balancing,...). The first aspect ensures that StarSs applications are portable and the second reduces the turnaround time for code development. The high-level interface of StarSs nevertheless translates to good performance for a wide variety of applications. To this end the run-time libraries incorporate techniques that e.g. improve temporal locality, increase the degree of parallelism, pipeline tasks, cache data, and so forth.

### 2.4 GPU Architecture Consideration

We summarize some of the important aspects of the GPU architecture and its programmability that is pertinent to the efficient use of GPU kernels and structuring the flow of the integral histogram computation for large images.

**Massive parallelism and programmability** NVIDIA GPUs consist of several Streaming Multiprocessors (SMs), each containing a set of in-order cores. In the Fermi architecture, each SM comprises either 32 or 48 cores. For example, the Tesla C2070 card consists of fourteen 32-core SMs, for a total of 448 cores. The advent of the Compute Unified Device Architecture (CUDA) has greatly improved the programmability of NVIDIA GPUs. With CUDA, the computation is organized in a hierarchical fashion, wherein threads are grouped into thread blocks. Each thread block is mapped onto a different SM, whereas different threads are mapped to cores and executed in SIMD units, called warps. Threads within the same block can communicate using shared memory, whereas threads within different thread blocks are fully independent. Therefore, CUDA exposes to the programmer two degrees of parallelism: fine-grained parallelism within a thread block and coarse-grained parallelism across multiple thread blocks. GPU utilization is maximized when threads belonging to the same warp do not present divergent control flows, and when the kernel launch configuration (number of threads and thread blocks) is such to fully utilize the underlying cores.

**GPU memory hierarchy** GPUs have a heterogeneous memory organization consisting of high latency off-chip global memory, low latency read-only constant memory (which resides off-chip but is cached), low-latency on-chip read-write shared memory, and texture memory. GPUs adopting the Fermi architecture, such as those used in this work, are also equipped with a two-level cache hierarchy. Judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieve good performances.

The global memory can be accessed via 32-, 64- or 128- bytes transactions. Multiple memory accesses to contiguous memory locations can be automatically coalesced into a single memory transaction: memory coalescing is fundamental to optimize the memory bandwidth utilization. Further, context-switch among threads can be used to hide high latency global memory accesses. Finally, the use of shared memory and caches can be used to reduce the accesses to global memory.

The shared memory can be configured either as a software- or as a hardware-managed cache. The first configuration is typical of highly optimized code. In this case, within kernel functions, threads will first load the data from global memory to shared memory, then process the data into shared memory, and finally move the results into global memory. Since shared memory does not impose the same coalescing rules as global memory, it can allow efficient irregular access patterns. In order to achieve high bandwidth, shared memory is divided into equally sized banks. Memory requests directed to different banks can be served in parallel, whereas requests to the same bank are serialized. Therefore, avoiding bank conflicts is essential to optimize shared memory access.

## 3. METHODS, ASSUMPTIONS AND PROCEDURES

### 3.1 Integral Histogram Computation for Vision Applications

We define an image as a function $f$ over a two-dimensional Cartesian space $\mathcal{R}^2$ such that $\mathbf{x} \rightarrow f(\mathbf{x})$ for a pixel $\mathbf{x} \in \mathcal{R}^2$. The binning function $Q(f(\mathbf{x}), b)$ evaluates to 1 if $f(\mathbf{x}) \in b$ for the bin $b$, otherwise its value equals 0. For a set $B = \{b_0, b_{1,}, \ldots, b_n\}$ of intervals or bins the histogram $h$ evaluates the binning function $Q$ over the pixels in the domain of $f$:

$$h(b) = \sum_{\mathbf{x}} Q(f(\mathbf{x}), b) \quad b \in B \tag{1}$$

In general, IH associates a histogram with each pixel of the image. For a sequence of pixels $S = (\mathbf{x}^0, \mathbf{x}^1, \ldots, \mathbf{x}^p)$ and a subsequence $S_{\mathbf{x}^p} \subseteq S$ the integral histogram $H$ at $\mathbf{x}^p$ for a bin $b$ is defined as

$$H(\mathbf{x}^p, b) = \sum_{\mathbf{x} \in S_{\mathbf{x}^p}} Q(f(\mathbf{x}), b) \tag{2}$$

$S$ forms the *scan order*. This definition states that the value for a bin $b$ at a pixel $\mathbf{x}^p$ in $H$ can be found by applying the binning function to a subset of the pixels preceding $\mathbf{x}^p$ in the scan order. The computation essentially propagates pixel values through the image $f$. To emphasize the two-dimensional nature of $f$ we can identify the pixel $\mathbf{x}$ with its spatial coordinates $(i, j)$. This carries over to $H$ and $f$, that become $H(i, j, b)$ and $f(i, j)$ in this notation. We assume that $H$ and $f$ are respectively a vector and a scalar associated with a pixel, although in practice two separate data structures are used to represent the image and the integral histogram.

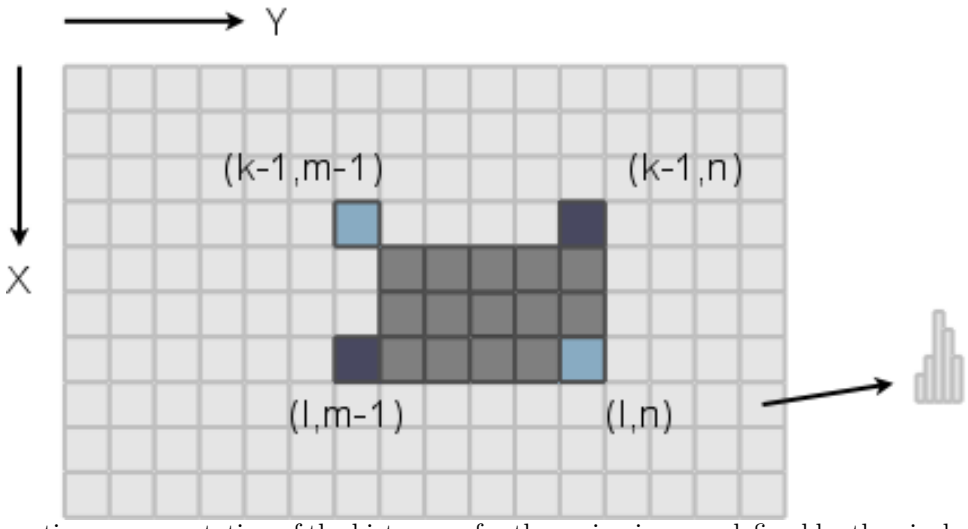



Figure 2: Intersection or computation of the histogram for the region in grey, defined by the pixels $\{(k, m), (k, n), (l, m), (l, n) | k < l, m < n\}$.

Specifically, IH uses an efficient scan order, that limits the number of visits to each pixel, such that the computation of the histogram for a rectangular region $T$ of $f$ becomes computationally inexpensive. In that context the computation of $H$, or **propagation**, precedes the computation of the histogram or **intersection**. We only consider scan orders that result in

$$H(i, j, b) = \sum_{x=0}^{i} \sum_{y=0}^{j} Q(f(x, y), b) \tag{3}$$

This means that $H$ at pixel $(i, j)$ summarizes the values of the pixels above and to the left of $(i, j)$. The intersection for the region (Figure 2) delimited by the points $\{(k, m), (k, n), (l, m), (l, n) | k < l, m < n\}$ then

reduces to a linear combination of four histograms from $H$:

$$\begin{aligned} h(T,b) =& H(k-1,m-1,b) + H(l,n,b) \\ & - H(k-1,n,b) - H(l,m-1,b) \end{aligned} \tag{4}$$

Propagation initializes $H(i,j,b)$ to 0 for all pixels $(i,j)$ and bins $b$. We describe two scan orders. A scan order, by definition, imposes a strict order on the pixels and histograms. But if the computation at an element $(i,j)$ requires only a proper subset of the histograms computed previously, the scan order can be relaxed to a partial order.

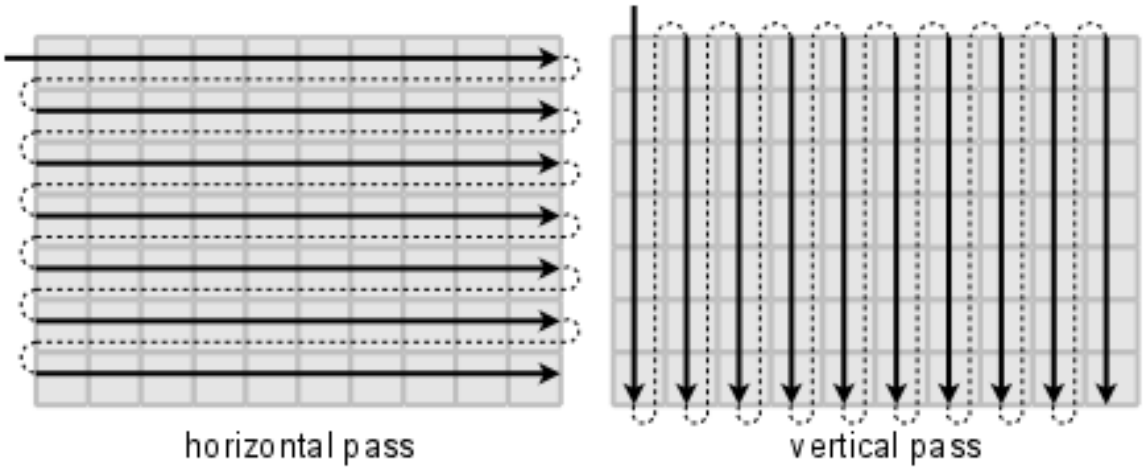



Figure 3: The two passes for the cross-weave scan.

The **cross-weave scan** requires two passes over $H$ as it processes each dimension separately (Figure 3). In the $Y$-direction it visits the pixels and histograms from left to right and top to bottom. At each element $(i,j)$ the algorithm propagates $H(i,j,b)$ to $(i,j+1)$ in order to compute $H(i,j+1,b)$. As a result the propagation in the $Y$-direction can advance in parallel for different rows, in spite of the strict order suggested by the horizontal pass. In the $X$-direction the cross-weave scan processes $H$ from top to bottom and left to right. The histogram at $H(i,j,b)$ passes on to pixel $(i+1,j)$ to compute $H(i+1,j,b)$. Now columns can be processed in parallel. The cross-weave scan thus allows the propagation for rows (step 1) and columns (step 2) to advance in parallel, while the operations in each row or column are strictly ordered:

$$\begin{aligned} &(1)\ H(i,j,b) = && 0 \\ &(2)\ H(i,j,b) = && Q(f(i,j),b) + H(i,j-1,b) \quad j>0 \\ &(3)\ H(i,j,b) = && H(i,j,b) + H(i-1,j,b) \quad i>0 \end{aligned}$$

The **wavefront scan** performs a single pass over the input image and orders the pixels according to the anti-diagonals, starting from the upper left corner down to the lower right corner (Figure 4). Propagation according to the wavefront scan computes $H(i,j,b)$ using $H(i,j-1,b), H(i-1,j,b), H(i-1,j-1,b)$. Applied recursively, the propagation at $(i,j)$ thus occurs after the algorithm has computed all $H(i-x,j-y,b), x=1,\ldots,i, y=1,\ldots,j$ above and to the left of $(i,j)$:

$$\begin{aligned} &(1)\ H(i,j,b) = && 0 \\ &(2)\ H(i,j,b) = && H(i-1,j,b) + H(i,j-1,b) \\ & && -H(i-1,j-1) + Q(f(i,j),b) \end{aligned}$$

Remark that, again, the wavefront scan does not strictly order the accesses to the elements. Its definition only requires that the computation of $H(i,j,b)$ precedes the computation of $\{H(i+i,j+j,b)\}, i,j>0$, as the latter uses the values computed by the former.

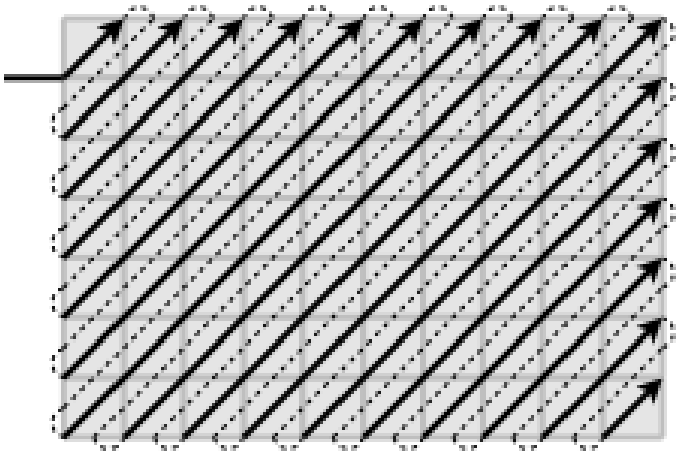


Figure 4: The wavefront scan.

A possible implementation of the wavefront scan advances an anti-diagonal wavefront over the input image. It first computes the histogram at the pixel $D_0 = \{(0,0)\}$, followed by the histograms at the pixels in $D_1 = \{(1,0),(1,1),(1,0)\}$,... until all the anti-diagonals have been visited. The histograms at each $D_i$ can be computed in parallel, because the histograms of all the (direct) upper and left neighbors have been computed in $D_{i-1}$. Alternatively an implementation can process the pixels from left to right and from top to bottom, under the restriction that only one element can be processed at a time.

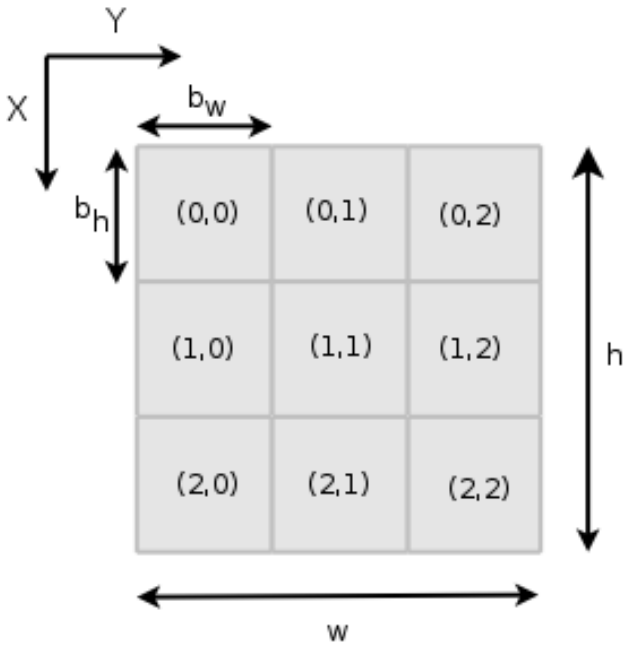



Figure 5: Block data layout for a $w \times h$ image and integral histogram. Each tile contains $b_w \times b_h$ pixels or histograms.

### 3.2 Parallel Integral Histogram in StarSs

The cross-weave nor the wavefront scan impose a strict order on the computations (Section 3.1), hence IH can be parallelized. To this end we formulate a tiled or blocked[41] version. Blocking subdivides the image $f$ and the integral histogram $H$, and naturally divides the computation into smaller units or *tasks*. This design is scalable and general, and readily adapts to different target architectures. The implementation in StarSs expresses the

tiled algorithm in a traditional, sequential programming language such as C. At run-time StarSs tracks read and write accesses to the tiles, derives the dependencies between the tasks performing the accesses and schedules tasks to the parallel resources (Section 2.3).

Figure 5 illustrates the tiled or block data layout for $f$ and $H$. We preserve the usual coordinate system, but now a coordinate pair identifies a tile instead of an individual pixel or histogram. An image of dimensions $w \times h$ defines $H$ with $(w \times h) \times b_c$ bins, with $b_c$ the number of bins in a histogram. It admits a division into blocks of $b_w \times b_h$ pixels, whereas $H$ can be decomposed in tiles of $b_w \times b_h$ histograms. Conversely, each tile of $H$ holds $b_w \times b_h \times b_c$ bins. The original image can be padded to eliminate boundary conditions. In the block data layout $f$ as well as $H$ consist of $w_B \times h_B$ blocks with $w_B = \lceil w/b_w \rceil$ and $h_B = \lceil h/b_h \rceil$. We distinguish between blocks and individual elements using a slightly different notation. Blocks $f_{i,j}$ and $H_{i,j}$ correspond to the tiles at row $i$ and column $j$ in the block data layout of the image and the integral histogram, respectively, while $(i, j)$ designates a pixel or a histogram depending on the context.

We define the propagation on a single block as a *task*. For the cross-weave scan this results in two different types of tasks, one for the horizontal and one for the vertical pass. The wavefront scan uses only one type of task. There is a one-to-one map between tasks (of the same type) and blocks in $f$ or $H$. Task $t_{i,j}$ uses block $f_{i,j}$ in the propagation for a block $H_{i,j}$. Parallel computation then derives from concurrent execution of tasks without dependencies. For a GPU, intra-task parallelism can be achieved by dividing the computation of the elements of a block $H_{i,j}$ over the threads in a thread block (Section 4.1.4). On the Cell/B.E. and for SMP a parallel resource is single-threaded. Hence $H_{i,j}$ can be computed with a sequential version of the cross-weave or wavefront scan, restricted to a single block. In both scan orders the tasks access the pixels of $f_{i,j}$, and for each pixel $(x, y) \in f_{i,j}$ it updates the bins of the associated histogram $(x, y)$ in $H_{i,j}$. For example, the task for the wavefront scan in our implementation processes the pixels from left to right and from top to bottom as suggested in Section 3.1:

```
// im = image block
// ih = integral histogram block
for (int x=0;x<b_h;x++) {
  for (int y=0;y<b_w;y++) {
    int *hist=&(ih[x][y]);

    // 1) read pixel (x,y) of im
    // 2) find the bin b it belongs to
    // 3) find the histograms N,W and NW
    // of the pixels above, left and
    // above and left of (x,y), resp.
    for (int k=0;k<b_c;k++) {
      int incr=k==b?1:0;
      hist[k]=N[k]+W[k]-NW[k]+incr;
    }
  }
}
```

Similarly, the task for the horizontal pass of the cross-weave scan copies the histogram at $(x, y) \in H_{i,j}$ to the histogram at $(x, y + 1)$ and increments the bin $b$ for which $Q(f(x, y)) = 1$. The vertical scan adds the histogram at $(x - 1, y) \in H_{i,j}$ to the histogram at $(x, y)$. For both scan orders most of the computations in a tile $H_{i,j}$ use updated elements from $H_{i,j}$ as input. However, in the wavefront task, the computation of the histograms at $(0, y), y = 0, \ldots, b_w$ and $(x, 0), x = 0, \ldots, b_h$ for a block $H_{i,j}$ requires histograms that belong to $H_{i-1,j}, H_{i,j-1}$ and $H_{i-1,j-1}$. The horizontal pass of the cross-weave scan computes the elements $(x, 0)$ of $H_{i,j}$ using the histograms on the border of $H_{i,j-1}$ and the vertical pass requires the lower border of $H(i - 1, j)$ for the propagation of the $(0, y)$ in $H_{i,j}$. Propagation on blocks exhibits the same characteristics as propagation on individual elements. In a tiled formulation, IH propagates histograms from one block to another block at block boundaries. We restrict our attention to the wavefront scan from here on. The development of the algorithm for the cross-weave scan is analogous.

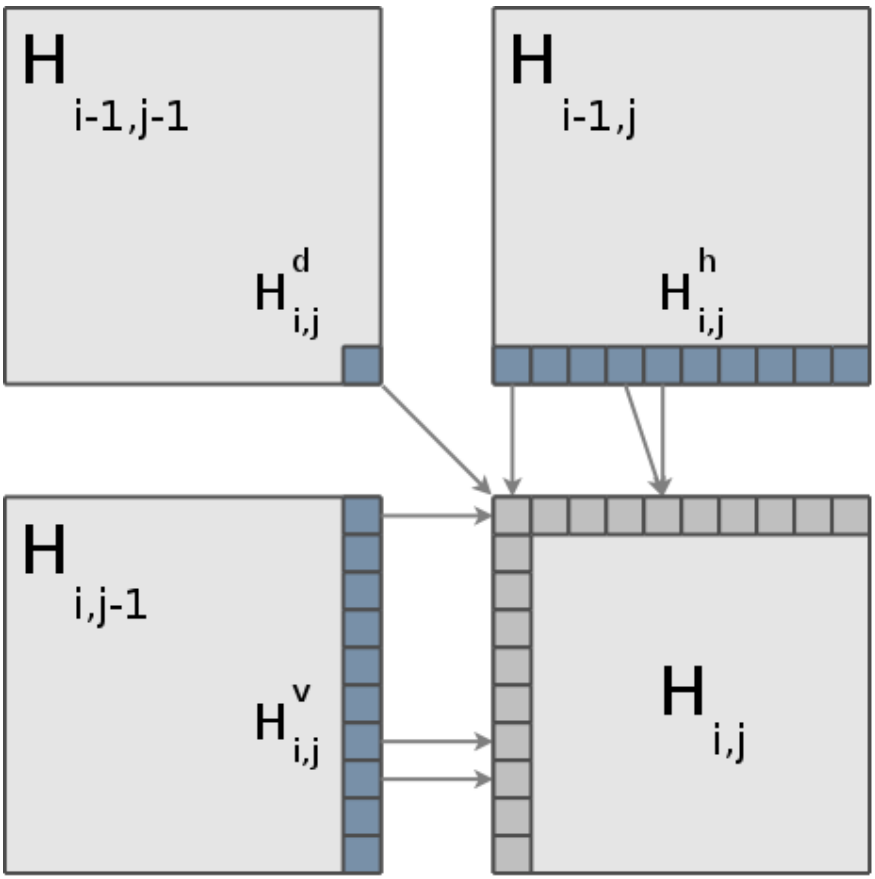


Figure 6: Inter-block dependencies for block $H_{i,j}$ with halos for propagation between tiles. The arrows specify the inter-block dependencies for some selected histograms on the borders of $H_{i,j}$.

The aforementioned dependence between $H_{i,j}$ stems from the data dependence between the elements they contain. Figure 6 identifies three sets or **halos** for a tile $H_{i,j}$, namely $H^d_{i,j}, H^h_{i,j}$ and $H^v_{i,j}$. The subscript identifies the tile whose edges require the histograms in these sets. The singleton $H^d_{i,j}$ contains the histogram $(b_h - 1, b_w - 1)$ of the diagonally opposite block $H_{i-1,j-1}$, while $H^v_{i,j} = \{(l, b_w - 1) \in H_{i,j-1} | l = 0, \ldots, b_h - 1\}$ and $H^h_{i,j} = \{(b_h - 1, l) \in H_{i-1,j} | l = 0, \ldots, b_w - 1\}$. Element $(0,0)$ of $H_{i,j}$ depends on $H^d_{i,j}$, elements $(x, 0), x = 0, \ldots, b_h - 1$ need $H^v_{i,j}$ and finally the propagation for elements $(0, y), y = 0, \ldots, b_w - 1$ uses $H^h_{i,j}$. The map between tasks and blocks defines task precedence via the data dependencies on the blocks: $t_{i,j}$ is eligible for execution if all tasks $t_{x,y}$ with $x < i, y < j$ have been computed. Or $t_{i,j}$ depends on $t_{i-1,j}$ via $H^h_{i,j}$, on $t_{i,j-1}$ via $H^v_{i,j}$ and on $t_{i-1,j-1}$ through $H^d_{i,j}$.

The halos correspond to definitions of sets of histograms, rather than effective data structures, that make the data dependencies between $H_{i,j}$ explicit. These blocks have fixed dimensions, while the propagation in IH extends over the entire input and spills from one block to the next. Our implementation replicates the halos and allocates separate physical buffers, as in Figure 7, which are considered blocks as well. This duplication allows us to express the data accesses by a task $t_{i,j}$ in terms of full blocks, instead of subsets of blocks: $t_{i,j}$ accepts as input arguments the block $f_{i,j}$ and the halos $H^h_{i,j}, H^v_{i,j}$ and $H^d_{i,j}$. Its output consists of $H_{i,j}$ and the halos $H^h_{i+1,j}, H^v_{i,j+1}$ and $H^d_{i+1,j+1}$. The latter in turn are read by $t_{i+1,j}, t_{i,j+1}$ and $t_{i+1,j+1}$. These chains of halo production and consumption establish the required data dependencies. The data dependencies as exposed via the task arguments completely define the task precedence, which simplifies and speeds up the dependence analysis in StarSs [*]. Figure 8 depicts the TDG for IH for a small image size for both scan orders. The cross-weave scan

[*]Although this is not a strict requirement.[35]

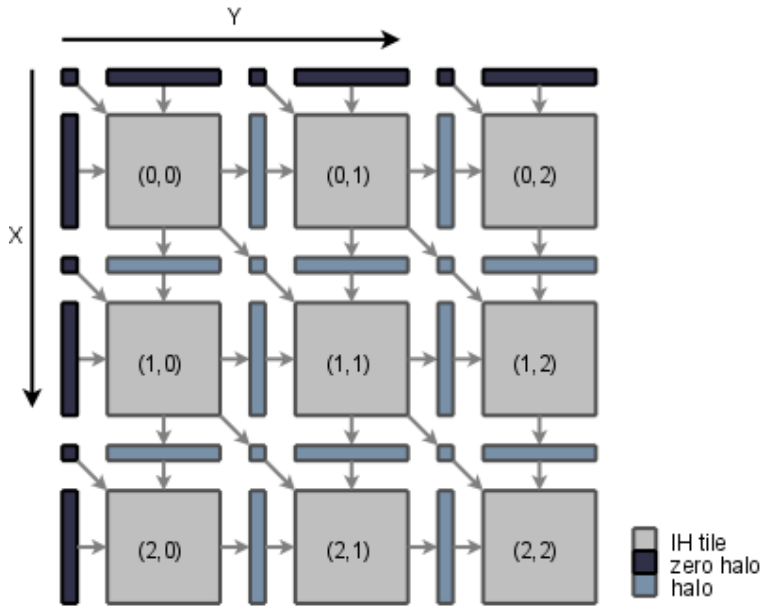


Figure 7: Block data layout for the integral histogram. Each block contains $b_w \times b_h$ histograms. Block borders are duplicated in the *halos* and serve to pass histograms to the neighboring blocks.

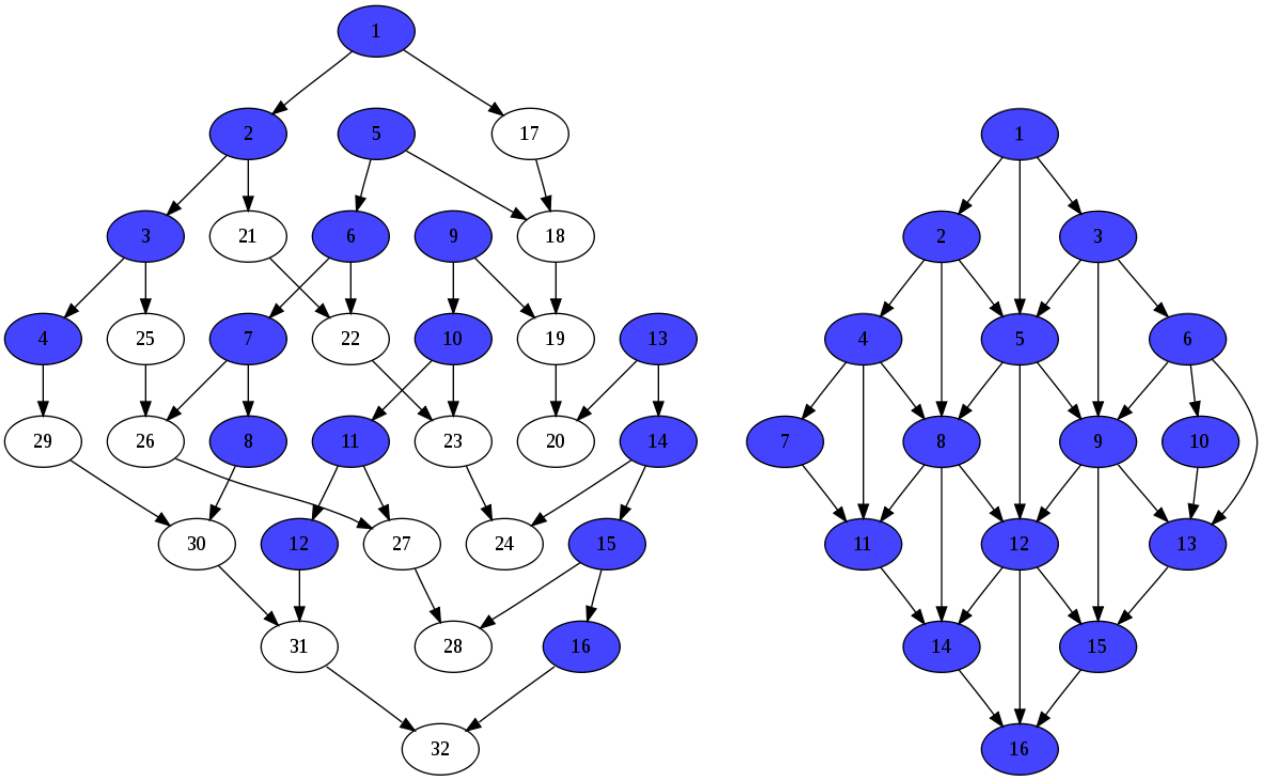
visits each tile $H_{i,j}$ twice, once during the horizontal pass and next in the vertical pass, and generates twice as many tasks as the wavefront scan.

The halos $H^v_{i,j}$ occupy an additional $h_B \times b_h \times w_B \times b_c$ bins, the halos $H^h_{i,j}$ take up $w_B \times b_w \times h_B \times b_c$ bins and the $H^d_{i,j}$ $h_B \times w_B \times b_c$ bins. These storage requirements can be reduced by noting that the horizontal halos can be recycled per row, the vertical halos per column and the diagonal halos per diagonal. The lifetime of $H^h_{i,j}$ ends before $H^h_{i+1,j}$ is produced because task $t_{i,j}$ executes and finishes before $t_{i+1,j}, \forall j = 0, \ldots, w_B - 1$. The task precedence for this application guarantees that the accesses to $H^h_{i,j}$ do not overlap in time for fixed $j$. Similar observations hold for the vertical and diagonal halos. Hence there is no reason to separate the input and output halos of a task: $H^h_{i,j}, H^v_{i,j}$ and $H^d_{i,j}$ can occupy the same memory as $H^h_{i+1,j}, H^v_{i,j+1}$ and $H^d_{i+1,j+1}$ respectively. With this reduction the horizontal halos occupy $w_B \times b_w \times b_c$ additional bins, the vertical halos $h_B \times b_h \times b_c$ bins and the diagonal halos $(w_B + h_B - 1) \times b_c$ bins. As a side-effect the task can be defined with fewer parameters, because the halos that are read and written are the same. At run-time this translates to fewer arguments per StarSs task, less dependence analysis and less runtime overhead.

With the block data layout in place IH can be implemented in a straightforward manner in StarSs. The main function is a simple sequential description of the scan order. For the wavefront scan the code steps through the diagonals parallel to the minor diagonal of the blocked $H$. This generates the sequence of tasks $t_{0,0}$,$t_{1,0}$,$t_{0,1}$,$t_{2,0}$,$t_{1,1}$,$t_{0,2}$,... The StarSs library builds the TDG at execution time and schedules the tasks to the resources (Section 2.3). For SMP and the Cell/B.E. the task for the wavefront scan resorts to the sequential, left-to-right variant for propagation (Section 3.1). The main function for the cross-weave scan looks as follows:

```
// im[i][j] = image block (i,j)
// ih[i][j] = integral histogram block (i,j)
// vhalos[i] = vertical halo for row i
// hhalos[i] = horizontal halo for column i

int main(int argc,char *argv[]) {
...
// horizontal scan
for (int i=0; i<imheight; i++) {
```




(a) (b)

Figure 8: TDG for the tiled IH ($w_B = h_B = 4$) for (a) the cross-weave scan and (b) the wavefront scan. The tasks are numbered according to program order, which is identical to the scan order in our implementation in StarSs. The color of a node represents the task type.

```
  for (int j=0; j<imwidth; j++) {
    hscan(...,im[i][j],vhalos[i],ih[i][j]);
  }
}

// vertical scan
for (int j=0; j<imwidth; j++) {
  for(int i=0; i<imheight; i++) {
    vscan(..,hhalos[j],ih[i][j]);
  }
}
...
}
```

### 3.3 Reference Implementation on the Cell/B.E.

To evaluate the quality of IH in StarSs we implemented the wavefront scan directly on the Cell/B.E. using the same general design as in Section 3.2, albeit with a flat data layout (as opposed to a block data layout). The rows of a block do not occupy consecutive locations in main memory, but we process $f$ and $H$ by blocks: the SPEs use Direct Memory Access (DMA) lists to perform scatter and gather operations on the data. For fair comparison, we reuse the code of the StarSs tasks in this reference implementation.

The data dependencies for the wavefront scan are characteristic of its blocked formulation, not of a particular

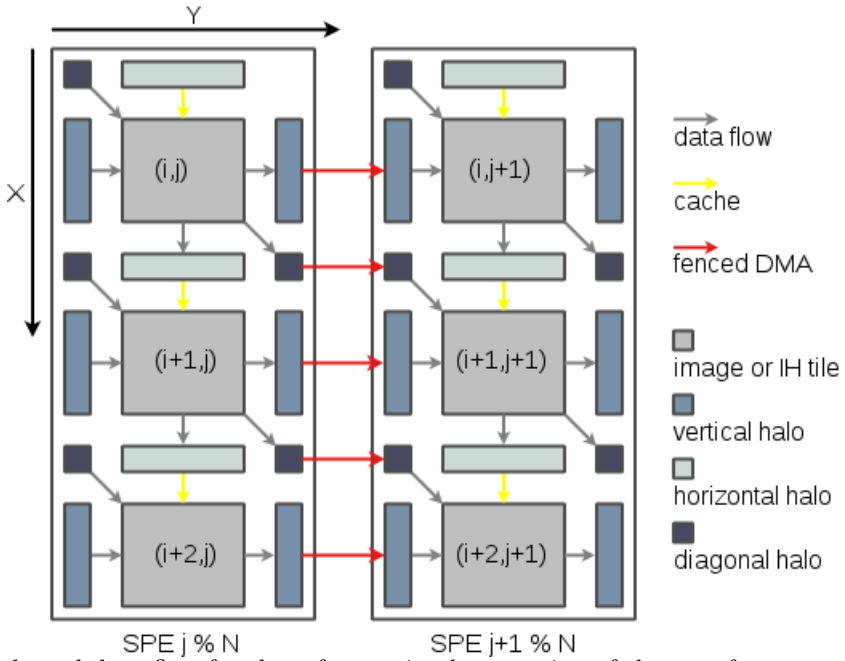


Figure 9: Schedule and data flow for the reference implementation of the wavefront scan on the Cell/B.E.

implementation. As the wavefront scan on the Cell/B.E. uses the same tasks as for the CellSs version, the task dependencies are identical (Figure 8(b)). The regularity of the task dependencies suggests that a static task schedule is feasible. We simply hard-code the trend we observed in the schedules from CellSs for the wavefront scan and assign task $t_{i,j}$ to SPE $j\%N$, with $N$ the number of available SPEs. Each (block) column in $f_{i,j}$ or $H_{i,j}$ is processed by the same SPE from top to bottom, in order, and the set of columns is divided over the SPEs in a round-robin fashion. As a result the halos ${H^h}_{i,j}$, $i = 0, \ldots, h_B - 1$ are accessed by the same SPE $j\%N$. They can be cached in the local store (LS) between tasks $t_{i,j}$ and $t_{i+1,j}$. From the schedule it further follows that the halos ${H^v}_{i,j+1}$ and ${H^d}_{i,j+1}$ have to be transferred from SPEs $j\%N$ to $(j+1)\%N$. We arrange for the former SPE to push the produced data to the latter via LS-to-LS DMA transfers. SPE $N-1$, the last one in the pipeline of SPEs, copies the vertical and diagonal halos to main memory. To close the cycle, SPE 0 transfers those halos from main memory to its LS when it processes the corresponding tiles. For the remainder of the report we drop the modulo notation, as the distribution of tasks to SPEs should be clear.

The stand-alone implementation must enforce the proposed schedule, which requires synchronization. We do not use a scheduler for this purpose, but rather implement a distributed mechanism with tokens that piggy-back onto the DMA transfers of blocks between SPEs. Remark that the schedule defines a producer-consumer relation between SPE $j$ and $j+1$ (wrapping around from $N-1$ to 0). Hence SPE $j+1$ must postpone the execution of $t_{i,j+1}$ until SPE $j$ has produced the required halos $H^h_{i,j+1}$ and $H^d_{i,j+1}$. Reciprocally, SPE $j$ must synchronize with $j+1$ in order not to flood the consumer with halos and exceed the limited storage of the LS. We conclude that the SPEs must work in lock-step.

The task dependencies dictate that $t_{i,j}$ must precede $t_{i+1,j}, t_{i,j+1}$ and $t_{i+1,j+1}$. The schedule trivially settles the first dependency. For the last two dependencies we arrange an explicit means of synchronization that takes advantage of the properties of the DMA transfers between each pair of SPEs. If the dependency $t_{i,j} \rightarrow t_{i,j+1}$ (on SPE $j$ and $(j+1)$ respectively) is satisfied, then so is $t_{i,j} \rightarrow t_{i+1,j+1}$, because $t_{i,j+1}$ precedes $t_{i+1,j+1}$ on SPE $(j+1)$ per definition of the schedule. We can therefore concentrate on the synchronization for the transfer of the vertical halos between a pair of SPEs to enforce the correct task precedence. We synchronize SPE $(j+1)$, as a consumer of $H^v_{i,j+1}$, with the producer, SPE $j$, by transferring a token together with each $H^v_{i,j+1}$. After $t_{i,j}$ finishes, SPE $j$ starts the DMA transfer of $H^v_{i,j+1}$ to SPE $j+1$ with tag $t$, followed by the transfer of a token $T_p$ to SPE $j+1$ using a DMA fence and the same tag $t$. SPE $j+1$ in turn delays the execution of $t_{i,j+1}$ until it detects the arrival of $T_p$. The fence attribute associated with the DMA transfer of $T_p$ guarantees that preceding

DMA transfers with the same tag are ordered with respect to SPE $j+1$. When SPE $j+1$ detects the arrival of $T_p$, $H^v_{i,j+1}$ will be present in its LS. For the wrap-around case, between SPE $N-1$ and SPE 0, the detection of $T_p$ by SPE 0 signals that the DMA transfer of the vertical halo from the LS of SPE $N-1$ to main memory has finished. The consumer, SPE 0, can then set up a DMA transfer to bring the halo to its LS.

The same mechanism prevents that an SPE produces halos faster than its neighbor is able to consume. "Faster" in this context means that the producer uses up all the available buffers in the LS of the consumer, before the latter consumes the halos kept in the buffers. SPE $j$ cannot advance unrestrictedly and flood the limited storage in SPE $j+1$ with vertical and diagonal halos. At some point SPE $j$ much check whether it can reuse buffers in the LS of SPE $j+1$ to store the halos it produces. To this end SPE $j+1$ transfers a token $T_c$ to SPE $j$. SPE $j$ uses the value of $T_c$ to check the amount of available buffers in SPE $j+1$. A buffer $b$ becomes available again when the halo it contains can be overwritten, i.e. when it has been successfully transferred from SPE $j+1$ to SPE $j+2$. After SPE $j+1$ starts the DMA transfer of $b$ to SPE $j+2$ with tag $t$, it reuses $t$ for a fenced DMA transfer of $T_c$ to SPE $j$. When SPE $j$ detects $T_c$ in its LS, it knows that the DMA transfer from $b$ to the LS of SPE $j+2$ has finished, and that it can overwrite $b$ with a new halo.

In summary, SPE $j$ must wait for $T_p$ from SPE $j-1$ before executing the associated task and for $T_c$ in order not to overwrite the corresponding buffer in SPE $j+1$. This type of tight synchronization is very sensitive to delays and overly restricts the computation and interaction between SPEs. Instead of providing one single LS buffer for the halos, the image tile or the integral histogram tile, an SPE creates multiple buffers for each. And instead of using dimensionless tokens, our implementation uses arrays of counters to implement $T_p$ and $T_c$. These provisions enable multi-buffering and relax the lock-step between the SPEs. The details are out of scope for this report, but hopefully Figure 10 illustrates the idea clearly.

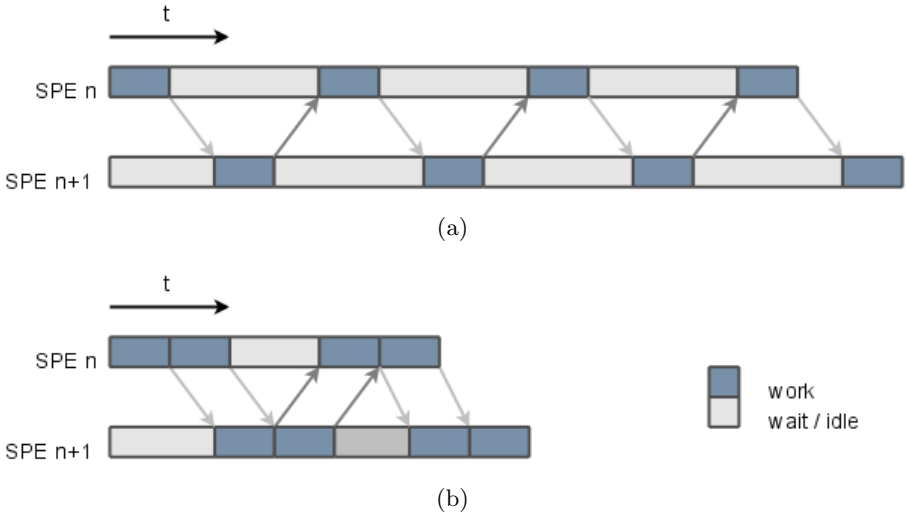



Figure 10: (a) Tight lock-step between SPEs $n$ and $n+1$ due to the presence of one output buffer for the produced halos. (b) The synchronization between the SPEs becomes less stringent by providing two output buffers. As a result the processing of the four blocks speeds up.

In Figure 10(a) SPE $n$ disposes of a single output buffer per halo, which hold the produced halos. SPE $n$ must wait for the acknowledgement ($T_c$) from SPE $n+1$ before it advances and overwrites these buffers, as their contents must have reached SPE $n+1$. This results in a tight lock-step between the SPEs involved and SPE cycles are wasted waiting for the halos to arrive or clear the LS. In Figure 10(b) the SPEs have two output buffers per halo. SPE $n$ can then execute two tasks and produce two sets of halos before it has to synchronize with SPE $n+1$, waiting for a buffer to free up. The additional buffer and the more refined synchronization protocol give the SPEs more freedom to advance before having to synchronize with its neighbors. The resulting decoupling reduces the time spent in synchronization and hence improves performance.

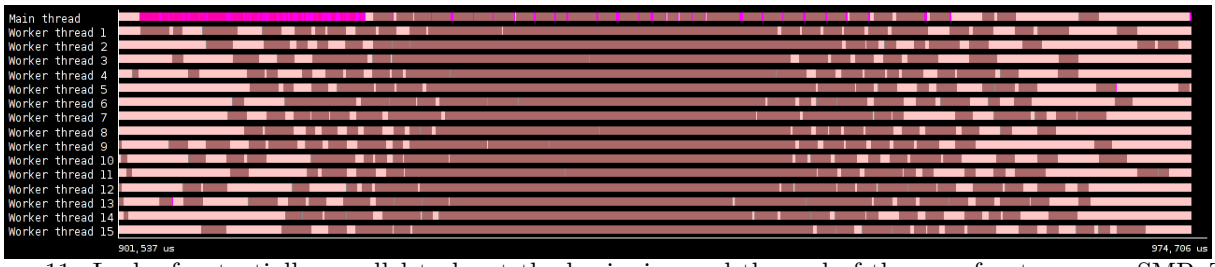


Figure 11: Lack of potentially parallel tasks at the beginning and the end of the wavefront scan on SMP. The horizontal axis represents execution time and an entry on the vertical axis corresponds to a thread. The dark phases mark task execution whereas the threads idle during the lighter phases. In this Paraver trace we clearly distinguish a computation-intensive middle part centered between two regions where the threads are less active.

### 3.4 GPU Kernel Optimization for Integral Histogram

In this section we describe the details of the GPU integral histogram implementation. We first describe the integral histogram data structure and its layout in GPU memory and then present different optimization strategies. In our first implementations,[7] we reuse existing parallel kernels from the NVIDIA Software Development Kit (SDK); we refer to these as generic kernels. We point out the limitations of such an approach, and progressively refine our implementation in order to better utilize the architectural features of the GPU. This leads to the evolution of four techniques to compute the integral histogram on GPUs that trade-off productivity with efficiency and a discussion of how the performance of the proposed implementations reflect their utilization of the underlying hardware. The first three implementations perform cumulative sums on row and column histograms in a cross-weave (CW) fashion, whereas the fourth one performs a wavefront (WF) scan.

#### 3.4.1 GPU Aware Data Structure Design

An image with dimensions $h \times w$ produces an integral histogram tensor of dimensions $b \times h \times w$, where $b$ is the number of bins in the histogram. This tensor can be represented as a 3-D array, which in turn can be mapped onto a 1-D row major ordered array as shown in Figure 12. It is well known that the PCI-Express connecting CPU and GPU is best utilized by performing a single large data transfer rather than many small data transfers. Therefore, whenever the 1-D array representing the integral histogram fits in the available GPU global memory, we transfer it between GPU and CPU using a single memory transaction. The computation of larger integral histograms is tiled along the bin-direction and distributed between available GPUs: portions of the 1-D array corresponding to the maximum number of bins that fit the GPU capacity are transferred between GPU and CPU in single transaction units. For all the considered image sizes a single bin fits the GPU memory; however, our implementation can be easily extended to images exceeding the GPU capacity by tiling the computation also column-wise. Finally, we experimentally verified that initializing the integral histogram on GPU is more efficient than initializing it on CPU and then transferring it from CPU to GPU. Therefore, in all our GPU implementations, we initially transfer the image from CPU to GPU, then initialize and compute the integral histogram on GPU, and finally transfer it back from GPU to CPU.

#### 3.4.2 GPU Parallelization Using Parallel Prefix-Sum (Exclusive Scan)

One basic pattern in parallel computing is the use of independent concurrently executing tasks. The recursive sequential Algorithm 1 is a poor approach to parallelize since row $(r + 1)$ cannot be executed until row $r$ is completed, with only intra-row parallelization. The cross-weave scan mode (Fig. 3), enables cumulative sum tasks over rows (or columns) to be scheduled and executed independently allowing for inter-row and column parallelization. The GPU Integral Histogram using Multiple Scan-Transpose-Scan (GIH-Multi-STS) is shown in Algorithm 2. This approach combines cross-weave scan mode with an efficient parallel prefix sum operation and an efficient 2-D transpose kernel. The SDK implementation of *all-prefix-sums* operation using the CUDA programming model is described by Harris, *et al.*[1] We apply prefix-sums to the rows of the histogram bins (horizontal cumulative sums or prescan), then transpose the array and reapply the prescan to the rows to obtain the integral histograms at each pixel.

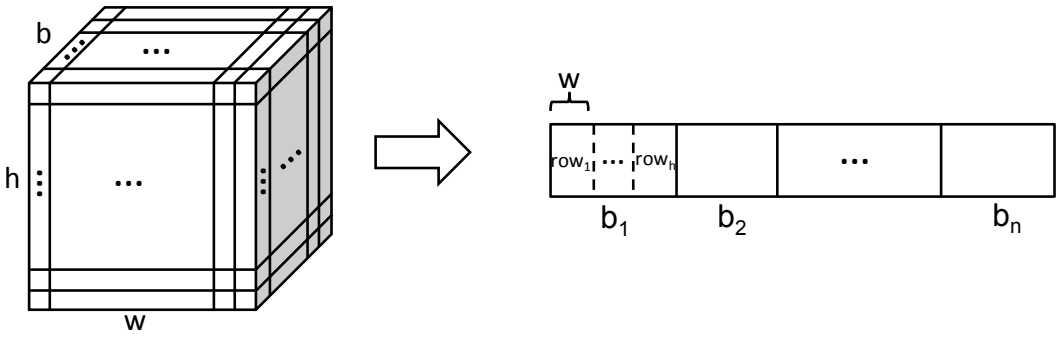



Figure 12: Integral histogram tensor represented as 3-D array data structure (left), and equivalent 1-D array mapping (right).

**Algorithm 1** GIH-Multi-STS: GPU Integral Histogram using Multiple Scan-Transpose-Scan

**Input :** Image $\mathbf{I}$ of size $h \times w$
**Output :** Integral histogram tensor $\mathbf{IH}$ of size $b \times h \times w$

**Initialize IH**
$\mathbf{IH} \leftarrow 0$
$\mathbf{IH}(\mathbf{I}(\mathbf{w}, \mathbf{h}), \mathbf{w}, \mathbf{h}) \leftarrow 1$
**for** z=1 to b **do**
  **for** x=1 to h **do**
    //horizontal cumulative sums (prescan, size of rows)
    $IH(x, y, z) \leftarrow IH(x, y, z) + IH(x, y-1, z)$
  **end for**
**end for**
**for** z=1 to b **do**
  //transpose the bin-specific integral histogram
  $IH^T(z) \leftarrow$ 2-D Transpose$(IH(z))$
**end for**
**for** z=1 to b **do**
  **for** y=1 to w **do**
    //vertical cumulative sums (prescan, size of columns)
    $IH(x, y, z) \leftarrow IH^T(y, x, z) + IH^T(y, x-1, z)$
  **end for**
**end for**

### 3.4.3 Parallel Prefix Sum Operation on the GPU

The core of the parallel integral histogram algorithm for GPUs is the parallel prefix sum algorithm.[1] The *all-prefix-sums* operation (also refered as a scan) applied to an array generates a new array where each element $k$ is the sum of all values preceding $k$ in the scan order. Given an array $[a_0, a_1, ..., a_{n-1}]$ the prefix-sum operation returns,

$$[0, a_0, (a_0 + a_1), ..., (a_0 + a_1 + ... + a_{n-2})] \tag{5}$$

The parallel prefix sum operation on the GPU consists of two phases: an *up-sweep* (or reduce) phase and a *down-sweep* phase (see Fig. 13). *Up-sweep* phase builds a balanced binary tree on the input data and performs one addition per node. Scanning is done from the leaves to the root. In the *down-sweep* phase the tree is traversed from root to the leaves and partial sums from the up-sweep phase are aggregated to obtain the final scanned (prefix summed) array. Prescan requires only $O(n)$ operations: $2 * (n-1)$ additions and $(n-1)$ swaps. The GPU-based prefix sum (prescan) operation moves data from CPU memory to off-chip global GPU memory then exploits the on-chip shared memory for each row operation.[1]

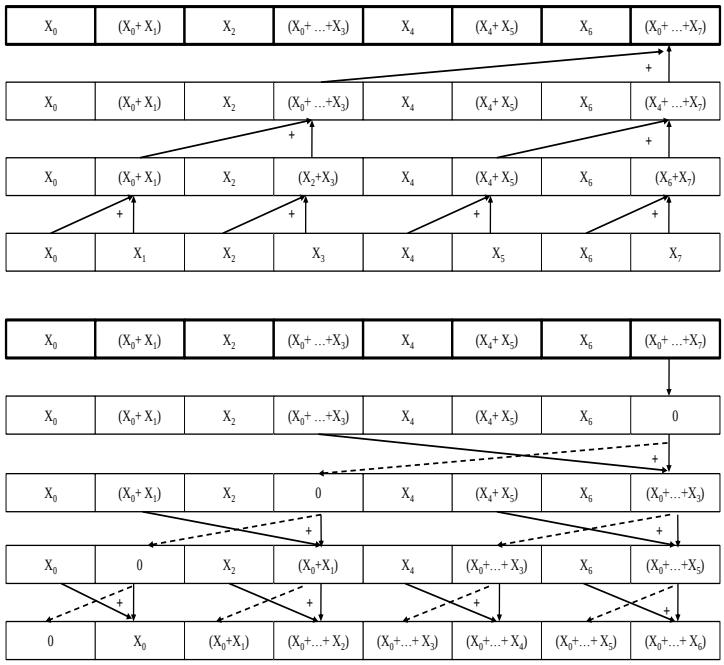


Figure 13: Parallel prefix sum operation, commonly known as exclusive scan or prescan.[1] Top: Up-sweep or reduce phase applied to an 8-element array. Bot: Down sweep phase.

### 3.5 GPU-based Transpose Kernel

The integral histogram computation requires two prescans over the data. First, a horizontal prescan that computes cumulative sums over rows of the data, followed by a second vertical prescan that computes cumulative sums over the columns of the first scan output. Taking the transpose of the horizontally prescanned image histogram, enables us to reapply the same (horizontal) prescan algorithm effectively on the columns of the data. We used the optimized transpose kernel described in[42] that uses zero bank conflict shared memory and guaranties that global reads and writes are coalesced. Figure 14 shows the data flow in the transpose kernel. A tile of size BLOCK_DIM $*$ BLOCK_DIM is written to the GPU shared memory into an array of size BLOCK_DIM$*$(BLOCK_DIM$+1$). This pads each row of the 2-D block in shared memory so that bank conflicts do not occur when threads address the array column-wise. Each transposed tile is written back to the GPU global memory to construct the full histogram transpose. The SDK 2-D transpose kernel needs to be launched from the host $b$ times in order to transpose the integral histogram tensor. In order to allow a single transpose operation, we transform the existing 2-D transpose kernel into a 3-D transpose kernel by using the bin offset in the indexing. The 3-D transpose kernel is launched using a 3-D grid of dimension $(b, w/\text{BLOCK\_DIM}, h/\text{BLOCK\_DIM})$, where BLOCK_DIM is the maximum number of banks in shared memory (32 for all graphics card used).

### 3.6 Data Structure and Implementation Strategy

An image with dimensions $h \times w$ produces an integral histogram tensor of dimensions $h \times w \times b$, where $b$ is the number of bins in the histogram. This tensor can be represented as a 3-D array which in turn can be mapped to an 1-D row major ordered array for efficient access as shown in Figure 12. Both implementations, GIH-Multi-STS and the improved GPU Integral Histogram using Single Scan-Transpose-Scan (GIH-Single-STS), start by prescanning each row. Since the maximum number of threads per block is 1024 and each thread processes two elements, each row can be divided into segments up to 2048 pixels. If the size of row is smaller than 2048 then the size of the thread block will be reduced to the $w/2$. The GIH-Multi-STS implementation uses the 3-D data structure. Exclusive prefix sum (prescan) kernel (see Section 3.4.3) is applied to the data one row at a time. This approach suffers from many kernel invocations in the horizontal/vertical scan and 2-D transpose phases, from little work per kernel and eventually GPU under-utilization (Algorithm 2 ). To reduce the total number of kernels invocations from $(w+h)b+b$ to only 3 invocations, the GIH-Single-STS implementation uses a 1-D row

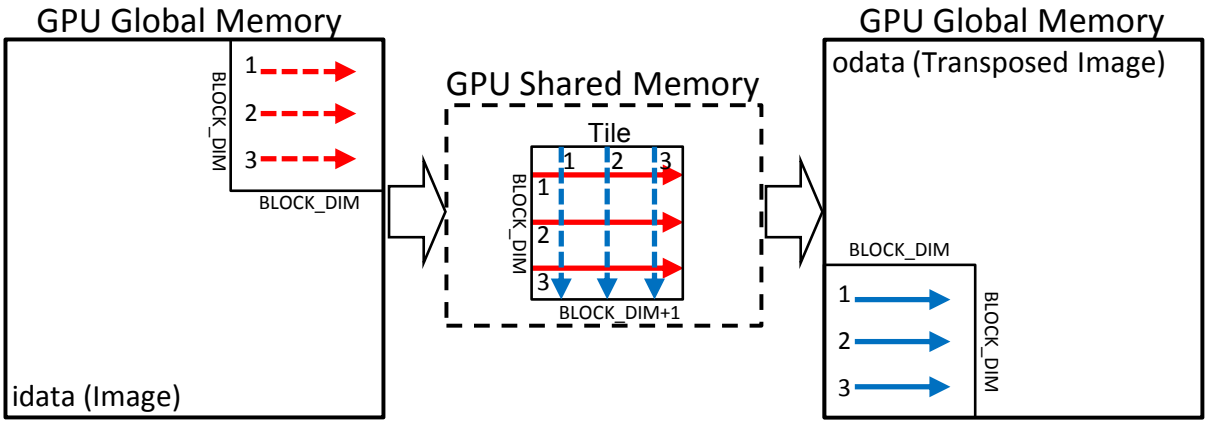



Figure 14: Data flow between GPU global memory and shared memory while computing the coalesced transpose kernel; stage 1 in red, stage 2 blue, reads are dashed lines, writes are solid lines.

ordered format array and launches the prescan kernel *once* using a 1-D grid of size $(b*h*w)/(2*\text{Num\_Threads})$. Padding is applied to shared memory addresses to avoid bank conflicts by adding an offset of 32 to each shared memory index. After prescanning each row (horizontal scan), the prescanned array is transposed to compute (column) cumulative sums in the second pass using a 3-D transpose kernel (Algorithm 3). We implemented and

**Algorithm 2** GIH-Single-STS: GPU Integral Histogram using Single Scan-Transpose-Scan

**Input :** Image $\mathbf{I}$ of size $h \times w$, number of bins $b$
**Output :** Integral histogram tensor $\mathbf{IH}$ of size $b \times h \times w$

**Initialize IH**
  $\mathbf{IH} \leftarrow 0$
  $\mathbf{IH}(\mathbf{I}(\mathbf{w},\mathbf{h}),\mathbf{w},\mathbf{h}) \leftarrow 1$
**for** all $b \times h$ blocks in parallel **do**
  //horizontal cumulative sums
  **Prescan**$(IH)$
**end for**
//transpose the histogram tensor
  $IH^T \leftarrow$ 3D_Transpose$(IH)$
**for** all $b \times w$ blocks in parallel **do**
  //vertical cumulative sums
  **Prescan**$(IH^T)$
**end for**

evaluated two parallel GPU integral histogram computation approaches: parallel GIH-Multi-STS, and parallel GIH-Single-STS and compared them to a sequential CPU-only implementation. Our experiments were conducted on a 2.0 GHz Quad Core Intel CPU (Core i7-2630QM) and two GPU cards: an NVIDIA Tesla C2070 and an NVIDIA GeForce GTX 460. The former is equipped with fourteen 32-core SMs and has about 5 GB of global memory, 48 kilobytes (KB) shared memory with compute capability 2.0. The latter consists of seven 48-core SM and is equipped with 1GB global memory, 48 KB shared memory with compute capability 2.1.

The parallel GIH-Multi-STS implementation exploits the work efficient prescan operation to calculate for each bin the cumulative sums of rows, one row at a time. Therefore, the scan kernel is launched $b \times h$ times for horizontal scan and $b \times w$ times for vertical scan. The efficient 2-D transpose kernel is launched $b$ times to transpose the integral histogram tensor after horizontal scan. The GIH-Multi-STS is based on many kernel invocations, each of them performing a small amount of work and therefore greatly under-utilizing the many-cores on the GPU. In addition, the all-prefix-sum kernel works very well only on very large array consisting of millions of elements. Therefore, we proposed the GIH-Single-STS to increase the amount of work performed by each kernel invocation and reduce the number of scan kernel invocations by a factor of $(h + w)b$. This can be easily

achieved by modifying the kernel configuration without rewriting the kernel code (array indices are derived from block and thread indices). Since the maximum number of threads per block is 1024 and each thread processes two elements, each row can be divided into segments up to 2048 pixels. If the size of row is smaller than 2048 then the size of the thread block will be reduced to the $w/2$ for horizontal scan and $h/2$ for vertical scan as well. Therefore, the number of blocks for horizontal scan will be $((b \times h \times w)/(2 \times threadblock))$. GIH-Single-STS also benefits from the modified 2-D transpose kernel which performs a single 3-D transpose operation by using the bin offset in the indexing. GIH-Single-STS is divided into three phases: a single horizontal scan, a 3-D transpose, and a vertical scan.

The initial implementation of GIH-Single-STS had several unnecessary data transfer between host and device after each phase. In the first implementation, the integral histogram tensor was being transferred to the GPU before invoking the kernel and then sent back to the CPU before launching the next kernel; these extra data transfers lead to reduced performance (referred to as GIH-Single-STS1). However, the GPU is specialized for compute-intensive, highly parallel computation and the overhead of communication between host and device cannot be hidden or double-buffered by non data-intensive kernels. In the improved GIH-Single-STS implementation, the integral histogram computations start after transferring the image to the GPU, complete the calculation of the integral histogram on the GPU then transfer the final integral histogram tensor back to the CPU, removing the extra communication overhead. In addition, the number of threads is automatically determined based on the image size to ensure maximum occupancy per kernel.

# 4. RESULTS AND DISCUSSION

## 4.1 Experimental Results for Integral Histograms Using StarSs

We implemented the cross-weave scan and the wavefront scan in StarSs according to the specifications in Section 3.3. This section contains performance results for various configurations on three architectures supported by StarSs. The implementation for the Cell/B.E. was tested on a *QS*22 blade (Section 4.1.1). The SMP version ran on an SGI Altix 4700 with 32 nodes (Section 4.1.3). Each node consists of two dual-core Itanium (Montecito) processors running at 1.6 GHz. For GPU we disposed of a system with two Intel Xeon E5649 6-Cores at 2.53 Ghz and two NVidia M2090 cards (Section 4.1.4). The code base of IH in StarSs for these three architectures is the same, although we specialized the task code for the Cell/B.E. and the GPU. The reported numbers are absolute and do not assess the quality of our implementation compared with traditional thread or stream models. To this end Section 4.1.2 compares IH in CellSs with a hand-coded version for the Cell/B.E. based on the description in Section 4.1.2. We compare the performance of both versions and try to quantify the programming effort in both cases. The default image size is $640 \times 480$ and the number of bins defaults to 16, 32 and 64.

As the cross-weave scan requires double the amount of tasks compared to the wavefront scan, we expect the latter to deliver better performance. The task dependencies for the cross-weave scan are less stringent than for the wavefront scan (Figure 8), but this does not outweigh the increase in task count and overhead. The task dependencies for the wavefront scan are likely to cause poor scalability. From the TDG we can see that the potential for parallel tasks is low at the beginning and towards the end of the execution. The execution of the first and the last task, for example, cannot overlap with the execution of other tasks. The sets of immediate neighbors do not exhibit significant parallelism either: parallelism opens up at the beginning of the execution and collapses again towards the end of the execution. This is a characteristic of the TDG of the wavefront scan, which is independent of the target platform or implementation, and is inherent to the formulation of the algorithm. Figure 11 depicts a Paraver[43,44] trace for the wavefront scan in SMPSs. It indeed demonstrates that ready tasks are too scarce to feed all the processors at the start and at the end. The middle part is rich with ready tasks, hence there is enough parallelism available to keep the resources from idling.

### 4.1.1 CellSs - StarSs for Cell/B.E.

Figure 15 summarizes the performance of the cross-weave scan and the wavefront scan in CellSs. Both versions accept the block size as an input argument. This implementation proves to be very practical, because the block size for best performance is not uniform across different bin counts ($b_c$). E.g. the cross-weave scan for 16 bins performs best for blocks of $28 \times 28$ elements, whereas 64 bins require smaller blocks of e.g. $13 \times 13$. In general, computations on an SPE must be well balanced, in the sense that code execution can seamlessly overlap the

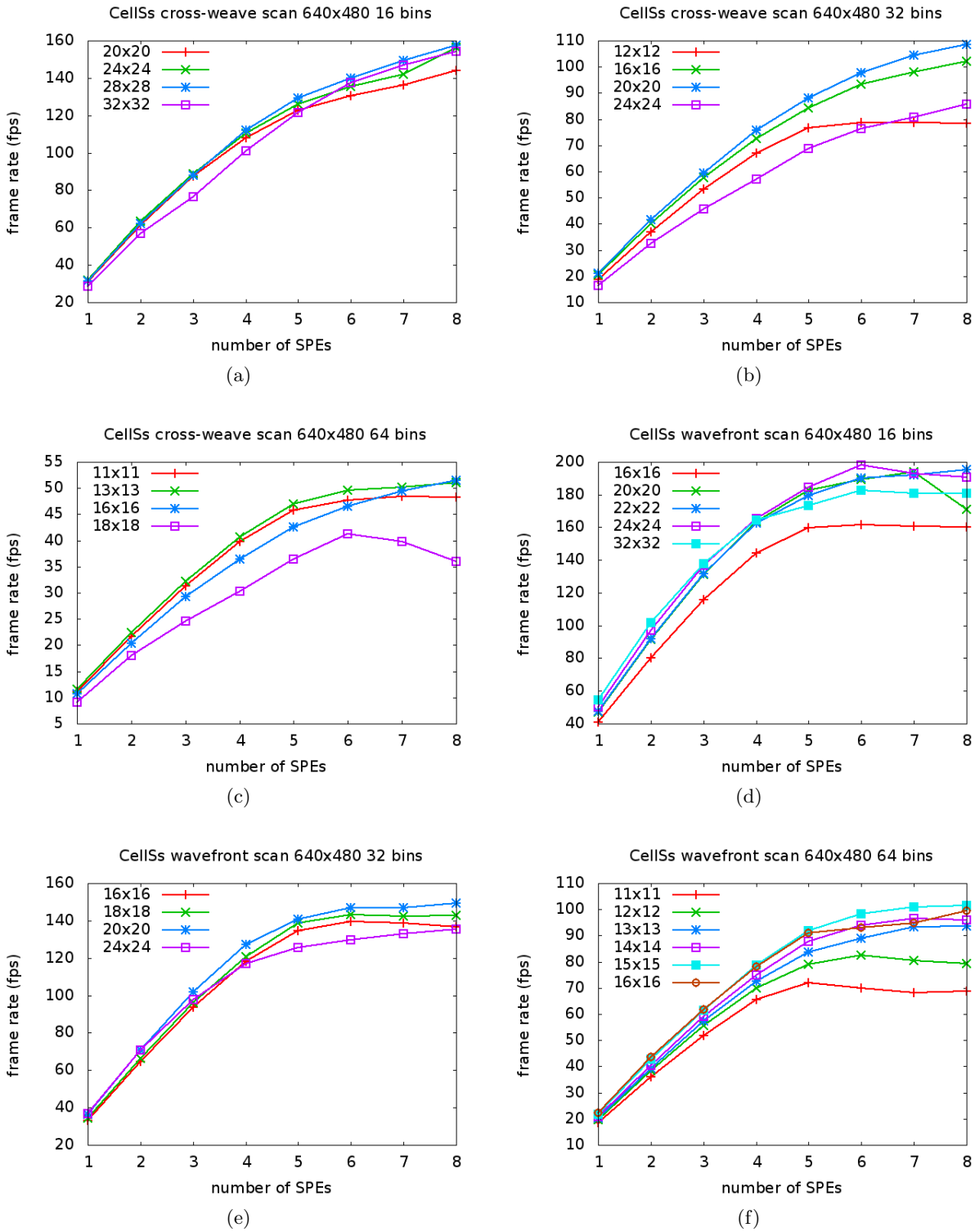



Figure 15: Performance of the cross-weave (a,b,c,d) and wavefront scan (e,f,g,h) in CellSs on an 640x480 image for different block sizes and different numbers of bins.

latency of the associated data transfers. This requirement relates to the granularity of tasks. If dynamic analysis (task creation, dependence analysis, scheduling, ...) takes place while the SPEs execute, run-time overhead influences the performance as well. Larger tiles divide the image or the integral histogram in fewer tiles, which results in less tasks and less overhead at execution time. The results in Figure 15 can be interpreted and understood as balancing granularity versus run-time overhead. The wavefront scan generates less tasks than the cross-weave scan for the same block size and consistently outperforms the latter. For both propagation methods the performance improves as the tasks become larger and fewer, until the size of the DMA transfers becomes prohibitive and hurts performance.

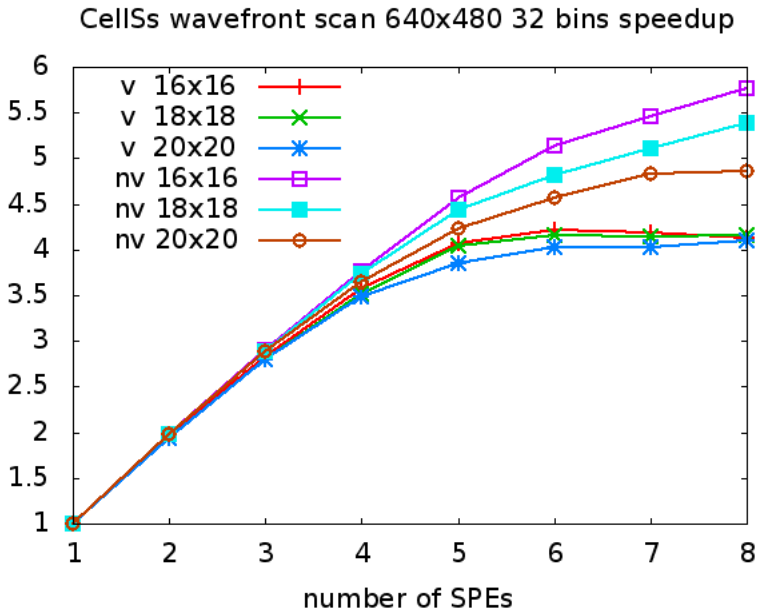



Figure 16: Speedup for the wavefront scan for 32 bins and different block sizes. One kernel has been vectorized ("v"), the other not ("nv").

The wavefront scan fails to scale well up to 8 SPEs. This is partly because of the very fine granularity of the vectorized tasks. For 32 bins, a single task executes in 18$\mu$s, 23$\mu$s and 28$\mu$s for blocks of $16 \times 16$, $18 \times 18$ and $20 \times 20$ elements, respectively. Naive kernels, which have not been vectorized, execute in 78$\mu$s, 98$\mu$s and 120$\mu$s for the same input. The latter tasks are coarser, succeed better at hiding the runtime overhead and ultimately scale better (Figure 16).

### 4.1.2 Cell/B.E. Intrinsics

The design of the wavefront scan for the Cell/B.E. described in Section 3.3 has three appealing properties compared to the implementation in CellSs:

- The schedule is fixed and exploits temporal locality.
- The SPEs synchronize loosely in a distributed fashion.
- The data traffic for the halos for the major part stays on the Element Interconnect Bus (EIB) and does not pass through main memory.

These qualities, together with the overlap of computation and communication, generally characterize a good design for the architecture in question. We implemented the algorithm without StarSs, using Synergistic Processing Unit (SPU) intrinsics and the thread Application Programming Interface (API) from the native Synergistic Processing Element (SPE) library. This stand-alone implementation serves as a yardstick for the implementation

| | CellSs | Cell/B.E. | Cell/B.E. (mem) |
|---|---|---|---|
| development time | 1 | 10 | 6 |
| code size | 1 | 6.8 | 5.9 |
| performance (fps) | | | |
| $b_c = 16$ | 222 | 405 | 350 |
| $b_c = 32$ | 175 | 280 | 208 |
| $b_c = 64$ | 87 | 151 | 134 |

Table 1: Comparison between IH in CellSs (Section 3.2), using the Cell SDK and LS-to-LS transfers (Section 3.3) and using the Cell SDK with LS-to-memory transfers. Development time and code size are normalized to the CellSs result. Performance is reported on a $640 \times 480$-image for $16, 32$ and $64$ bins.

in CellSs, as no other parallel implementation is available at the time of writing. To ensure a fair comparison, the CellSs and the Cell/B.E. version share the same kernel or task for the wavefront scan. These two different implementations specifically allow us to quantify the costs and benefits of a threaded, explicitly parallel model versus an implicit, data-dependence aware model. Portability is lost by using bare Cell/B.E. code, but we expect to improve performance by lowering the level of abstraction and reducing the run-time overhead. The performance improvement should be weighed against the cost of developing more intricate code. We also asses the contribution of the LS-to-LS DMA transfers to the overall performance, as this feature greatly increases the complexity of the design. Hence we consider a third version of the wavefront scan based off the design in Section 3.3 that passes the halos between SPEs via main memory, as the last and the first SPE in the proposed Cell/B.E. implementation.

Table 1 summarizes the results of this three-way comparison. The development time for the wavefront scan in CellSs is small compared to both Cell/B.E. versions. For the latter, the version that implements LS-to-LS DMA transfers took significantly longer to complete. The code sizes follow suit, although there is not much difference between the stand-alone implementations. IH in CellSs does without the thread creation, synchronization and communication mechanisms present in the other two versions, which keeps the code size small. As expected, the specialized implementations outrank the general, portable implementation in CellSs. The former achieves 55% to 65% of the performance of the specialized implementations.

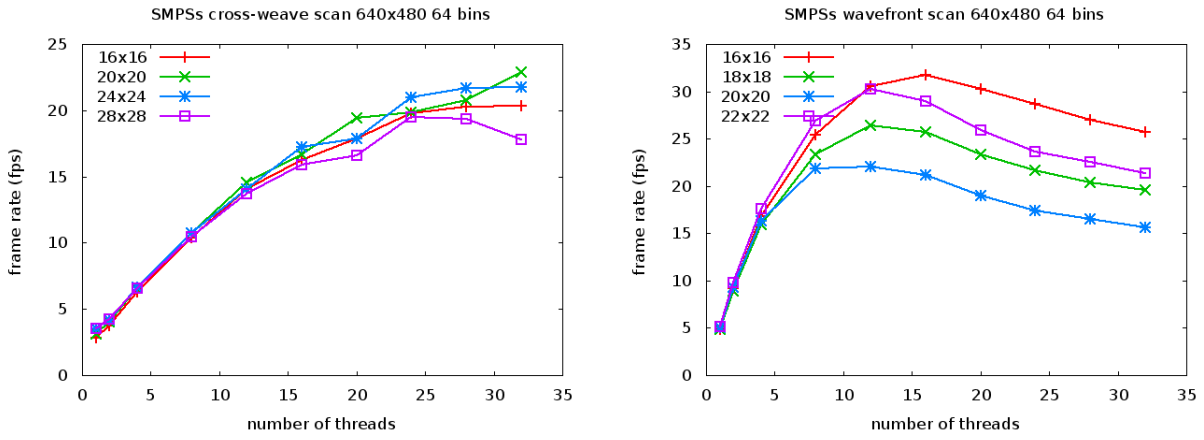
#### 4.1.3 StarSs for SMP

On SMP the cross-weave scan scales well for images of $640 \times 480$ and bins of 16, 32 and 64 bins. For these parameters the wavefront scan behaves well up to 16 threads, after which the performance starts to drop. Figure 17 illustrates this behavior for 64 bins. In Figure 18 we tentatively increased the granularity of the wavefront tasks by increasing the number of bins in the histogram. The higher accuracy of the histogram increases the duration of the task, but so does the amount of data transferred. The performance drop becomes less severe as the number of bins increases from 128 to 192 (top row of Figure 18), but the improvement is small. Ultimately the granularity of the wavefront task cannot be coarsened easily simply by increasing the tile size.

The lack of parallelism at the beginning and the end of the wavefront scan (Figures 8(b) and 11) exacerbates the performance drop. The cross-weave scan on the other hand has a TDG with more potential parallelism (Figure 8(a)) and hence scales better (Figure 17(a)). We validated this observation by increasing the image size to $1024 \times 768$ pixels. In combination with the previous tile sizes, the larger image generates more tasks, which decreases the fraction of the execution that suffers from low parallelism. The bottom row of Figure 18 displays the performance for the wavefront scan on a $1024 \times 768$ image. The performance drop for more than 16 threads has been relaxed, which confirms that the dependencies for the wavefront scan indeed affect the performance negatively.

#### 4.1.4 StarSs for GPUs

We did not specialize the wavefront task for GPUSs, the implementation of StarSs for GPU. Balancing such an inherently unbalanced computation (as argued in Section 4.1) on an architecture susceptible to thread divergence does not fit the scope of this report. The task code for the cross-weave scan, as we will see, is more amenable to the GPU.




(a) (b)

Figure 17: Performance of the cross-weave (a) and wavefront scan (b) on SMP for a 640x480 image for different block sizes and 64 bins.

Our specialization of the tasks of the cross-weave scan in C for CUDA maximizes the thread parallelism for the horizontal and vertical pass over a block $H_{i,j}$. Each row or column of $H_{i,j}$ can be processed independently (Section 3.1), and likewise, the bins in a histogram can be updated in parallel. Barring the block boundaries, the value of bin $b$ in element $(x, y)$ of $H_{i,j}$ depends on the value of bin $b$ of the histogram at $(x, y-1)$ of $H_{i,j}$ (and pixel $(x, y)$ of $f_{i,j}$) in the horizontal pass and on bin $b$ from $(x-1, y)$ at $H_{i,j}$ in the vertical pass. If we consider bins instead of histograms as basic elements, potential parallelism increases by a factor of $b_c$. In our specialization of the cross-weave task, the task of the horizontal pass (vertical pass) launches a single *thread block* with $b_h \times b_c$ ($b_w \times b_c$) threads. A thread updates a specific bin in each histogram of its row (column) (Figure 19). The GPUSs runtime manages the dependencies between these tasks and takes care of memory allocation on the device, stream management and data transfers between host and device.

The blocks are aligned to 128 bytes and we restrict bin sizes to multiples of 16, so the accesses to the $H_{i,j}$ in global memory are coalesced. Threads with consecutive IDs in a warp update consecutive bins. The number of thread blocks in a *thread grid* can be increased by grouping together $g$ rows or columns in a single thread block, instead of spanning all $b_h$ rows or $b_w$ columns. Each thread block then consists of just $b_c \times g$ threads. This additional blocking strategy only requires a specialization of the cross-weave tasks and leaves the rest of the source code untouched.

Table 2 lists the performance of various configurations of the cross-weave scan in GPUSs for different image sizes and various bin counts. Notice that performance improves as we increase the grouping factor $g$ and populate the thread grid with more thread blocks. The GPUSs scheduler can be configured to overlap computation and data transfers between host and device. This scheduling policy improves the frame rate for two GPUs moderately, e.g. for a $512 \times 512$ image and $g = 4$, the frame rate bumps from 113 fps to 121 fps. This experimental feature of GPUSs works fine as long as the amount of communication remains moderate (i.e. for smaller bin counts) and the grid topology is simple. Additionally, the Peripheral Component Interconnect (PCI) Express bus on our system provides limited bandwidth to main memory (8 GB/s). As the runtime starts using different streams and demanding more bandwidth, this sometimes causes performance to degrade for an already bandwidth-limited application as IH.

### 4.2 Summary for StarSs Integral Histogram

We described two parallel versions of the integral histogram (IH) based on a block data layout and proceeded to implement the cross-weave scan and the wavefront scan in StarSs. Each block corresponds to one or two types of tasks, respectively. The characteristic propagation of histograms in the original sequential formulation now appears at the block level. We unified the propagation of histograms at block boundaries by introducing *halos*.

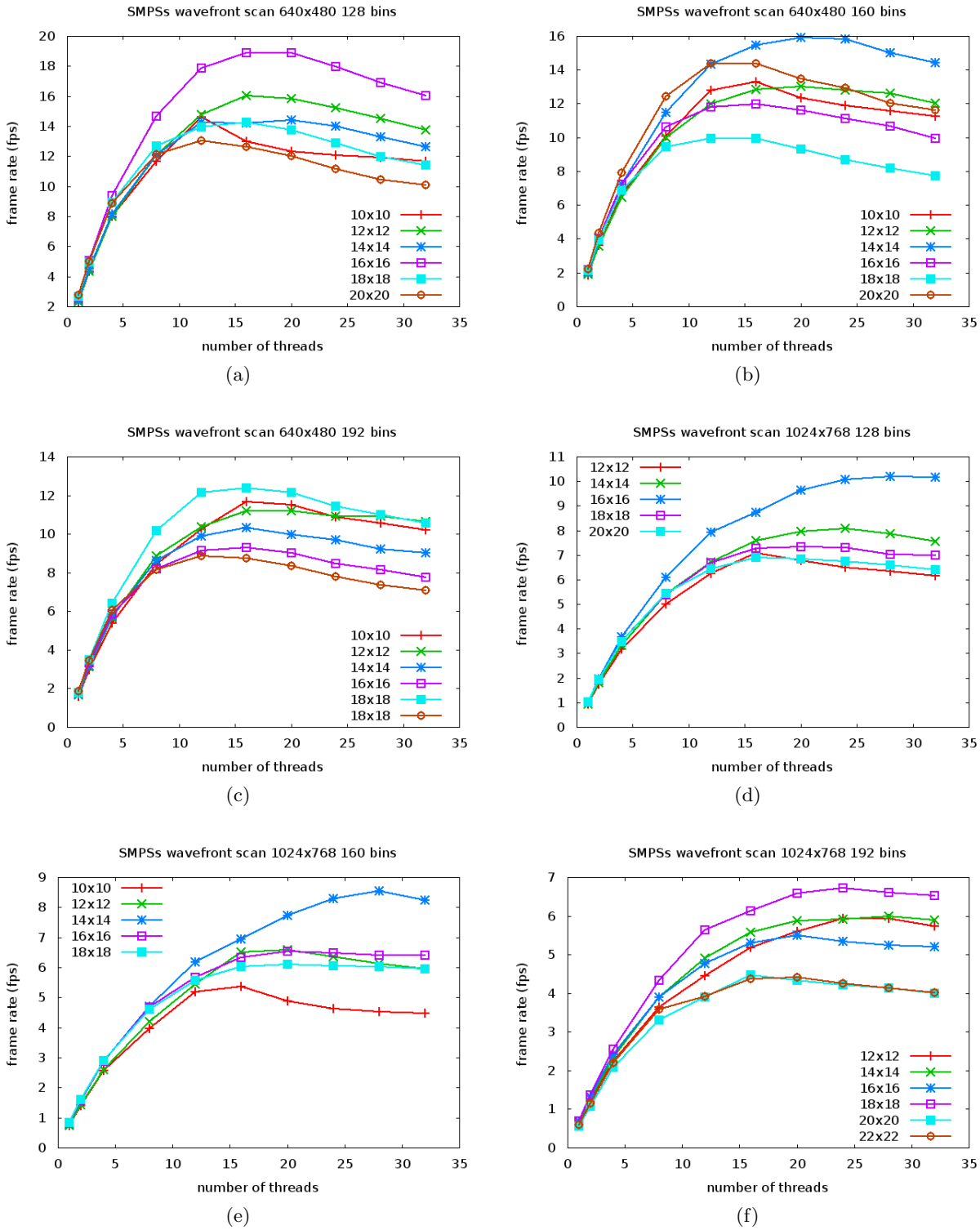



Figure 18: Performance of the wavefront scan on SMP for a $640 \times 480$ image (top row) and a $1024 \times 768$ image for different block sizes and 128,160 and 192 bins.

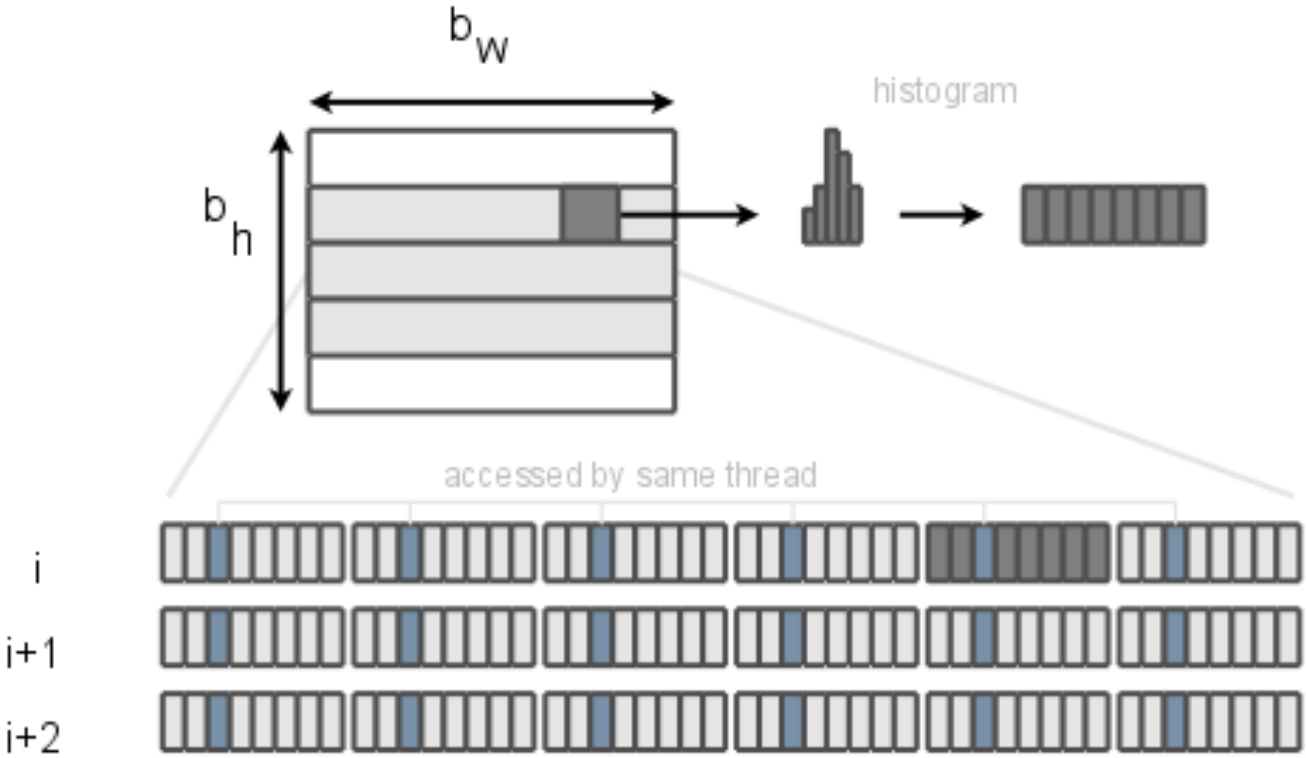



Figure 19: Memory layout and access pattern for the horizontal pass of the cross-weave scan on a block of the integral histogram. The block contains $b_w \times b_h$ histograms or $b_w \times b_h \times b_c$ individual bins. Each thread updates a single bin in one or more rows.

| $x \times y$ | $512 \times 512$ | | | | $640 \times 480$ | | | | $1024 \times 1024$ | | | | $1920 \times 1080$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_c$ | 16 | 32 | 64 | 128 | 16 | 32 | 64 | 128 | 16 | 32 | 64 | 128 | 16 | 32 | 64 | 128 |
| $g = 1$ | | | | | | | | | | | | | | | | |
| 1GPU | 60 | 25 | 8 | 2 | 47 | 21 | 1 | 0.2 | 15 | 6 | 1 | 0.2 | 8 | 6 | 3 | 2 |
| 2GPU | 74 | 33 | 12 | 3 | 60 | 27 | 3 | 0.4 | 19 | 8 | 2 | 0.4 | 10 | 6 | 3 | 2 |
| $g = 2$ | | | | | | | | | | | | | | | | |
| 1GPU | 120 | 72 | 37 | 20 | 57 | 39 | 23 | 14 | 35 | 21 | 11 | 6 | 8 | 6 | 4 | 2 |
| 2GPU | 125 | 70 | 38 | 20 | 69 | 42 | 23 | 14 | 37 | 21 | 10 | 6 | 10 | 6 | 4 | 2 |
| $g = 4$ | | | | | | | | | | | | | | | | |
| 1GPU | 133 | 70 | 37 | 22 | 111 | 65 | 35 | 18 | 39 | 21 | 11 | 5 | 19 | 10 | 5 | 3 |
| 2GPU | 113 | 71 | 36 | 20 | 108 | 63 | 34 | 17 | 37 | 20 | 10 | 5 | 18 | 10 | 5 | 3 |

Table 2: Framerate (fps) for images of $512 \times 512$, $640 \times 480$, $1024 \times 1024$, $1920 \times 1080$ elements and 16, 32, 64 and 128 bins, for 1 or 2 GPUs and using different grouping factors.

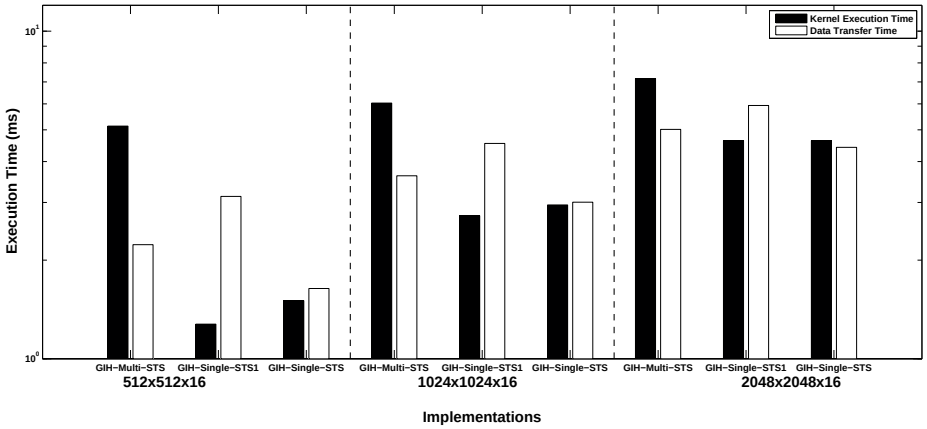



Figure 20: Kernel execution time versus data transfer time for different image sizes

This replication of data allows us to cleanly model the data flow of the application and speed up dependence analysis at run-time, but by no means is it mandatory.[35]

The code base is identical for the three architectures used in the study, namely SMP, the Cell/B.E. and NVidia CUDA. The implementations of StarSs (SMPSs, CellSs and GPUSs respectively) take care of scheduling, synchronization and data communication, which makes for very clean and compact code. The code for the tasks is compatible between SMPSs and CellSs, but GPUSs requires tasks to be written in C for CUDA, instead of standard C. The resulting changes to the code, however, are insignificant, consisting of one or two additional index calculations. Where applicable, we wrote specialized task code to take advantage of each architecture. Hence, the tasks process the elements of a block sequentially (SMPSs), in SIMD fashion (CellSs) or with multiple threads (GPUSs).

We like to think this demonstrates the portability of our solution and of dependence-aware, implicit programming models in general. The development time for these applications was short and the resulting code differs from standard C in but a few pragmas. However, a general solution that employs abstractions typically trades in efficiency for portability and simplicity. We analyzed the performance of IH in SMPSs, CellSs and GPUSs and included measurements for different image sizes and bin counts. Generally we observe good scalability for smaller bin counts and larger image sizes. The former avoids the large data communication associated with more detailed histograms, and the latter ensures the executions exhibit sufficient parallelism.

For the Cell/B.E. we additionally developed two optimized implementations, without StarSs, but tailored to the architecture. The comparison of these arguably good implementations with the implementation in CellSs succinctly illustrates the trade-off between performance and portability or programming cost.

Future work includes the development of parallel applications based on the implementation of IH described in this report. For example, we currently use our implementation to do Otsu-thresholding of surveillance images[45] produced by the flux Tensor[46] for moving object segmentation.

### 4.3 Experimental Results for Integral Histograms Using GPUs

Figure 20 shows the kernel execution time versus data transfer time for GIH-Multi-STS, GIH-Single-STS1 (implementation with extra data transfers) and GIH-Single-STS for different image sizes. We see that the GIH-Multi-STS is compute bound (that is, the kernel execution time is larger than the CPU to GPU data transfer time), this method under utilizes the GPU, whereas the GIH-Single-STS1 is data-transfer-bound. The results show that the data transfer time for GIH-Single-STS1 is on average five times worse than GIH-Single-STS. The final GIH-Single-STS implementation shows a balance between data transfer and kernel execution time (Fig. 20).

Figure 21 summarizes the frame rate performance of the two GPU implementations compared to the sequential CPU-only implementation. The frame rate is defined as the maximum number of images processed per second.

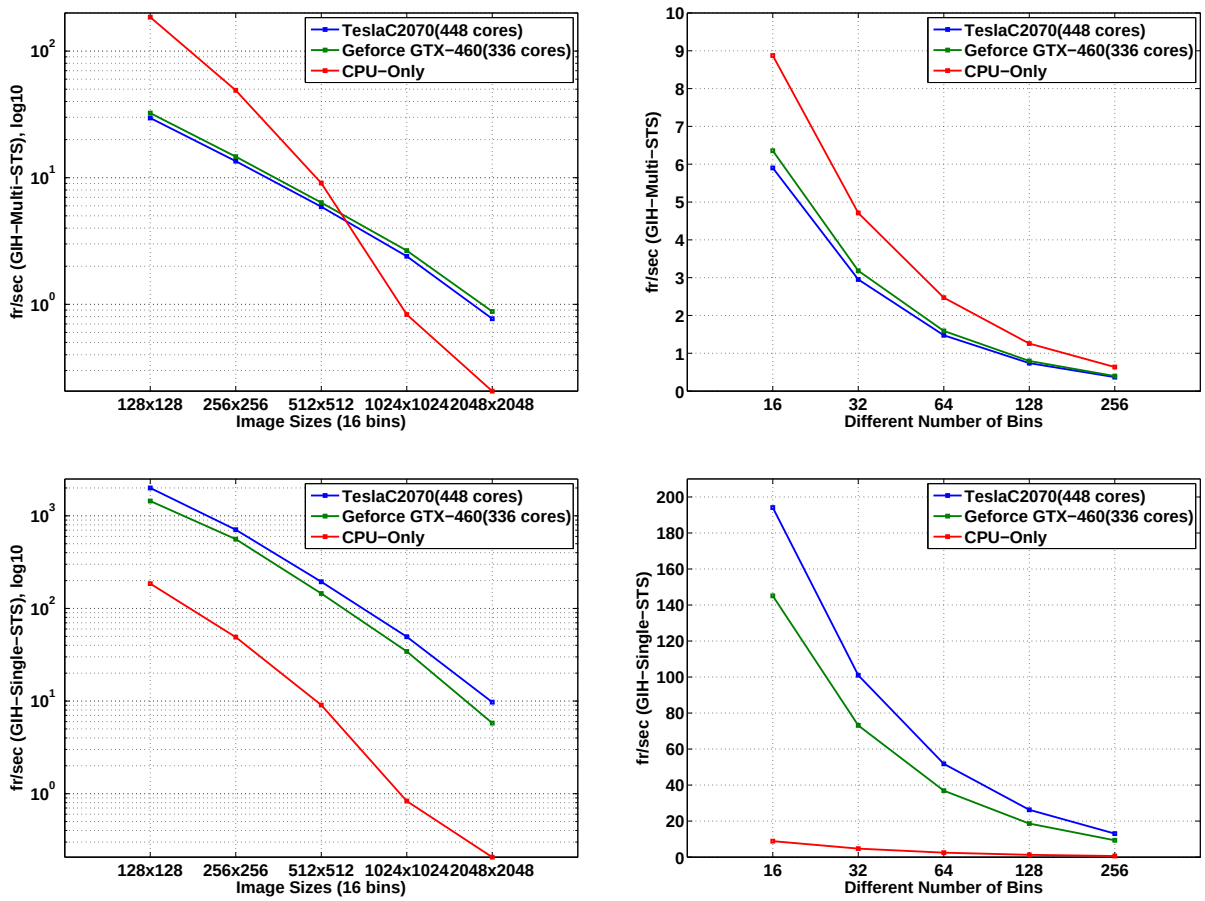



Figure 21: Frame rate of GIH-Multi-STS, GIH-Single-STS and CPU-only integral histogram implementations: (UL) GIH-Multi-STS frame rate for different image sizes, (UR) GIH-Multi-STS frame rate for different number of bins, (LL) GIH-Single-STS frame rate for different image sizes, (LR) GIH-Single-STS frame rate for different number of bins for 512x512 image size.

Since we use double buffering, the frame rate equals $1/(kernel\_execution\_time)$ for compute-bound cases, or $1/(data\_transfer\_time)$ for data-transfer-bound cases. Considering double buffering timing, our GIH-Single-STS achieves 194 fr/sec to compute 16-bin integral histograms for a $512 \times 512$ image and 94 fr/sec for $1K \times 1K$ image using the NVIDA Tesla C2070 GPU.

Figure 22 reports the speedup of our GPU implementations of the integral histogram compared to a sequential CPU implementation. The speedup takes into consideration the overlapping of computation and communication used by double buffering. The speedup of the improved GIH-Single-STS for a 16-bin integral histogram for a $1K \times 1K$ image is 60 times on an NVIDIA Tesla C2070 GPU and varies between 15 times to 25 times for a $512 \times 512$ image depending on the number of bins and the type of GPU.

Figures 23 shows feature maps for the target and search window with corresponding likelihood maps produced by the integral histogram-based likelihood estimation approach. Figure 24 shows sample tracking results and fused likelihood maps (using the integral histogram approach) for sample frames from an aerial wide area image sequence.

### 4.4 Summary for GPU Integral Histogram

We implemented two kernels for the integral histogram using the cross-weave scanning approach for GPU architectures, utilizing the CUDA programming model. The poor performance of the GIH-Multi-STS (*prescan*)

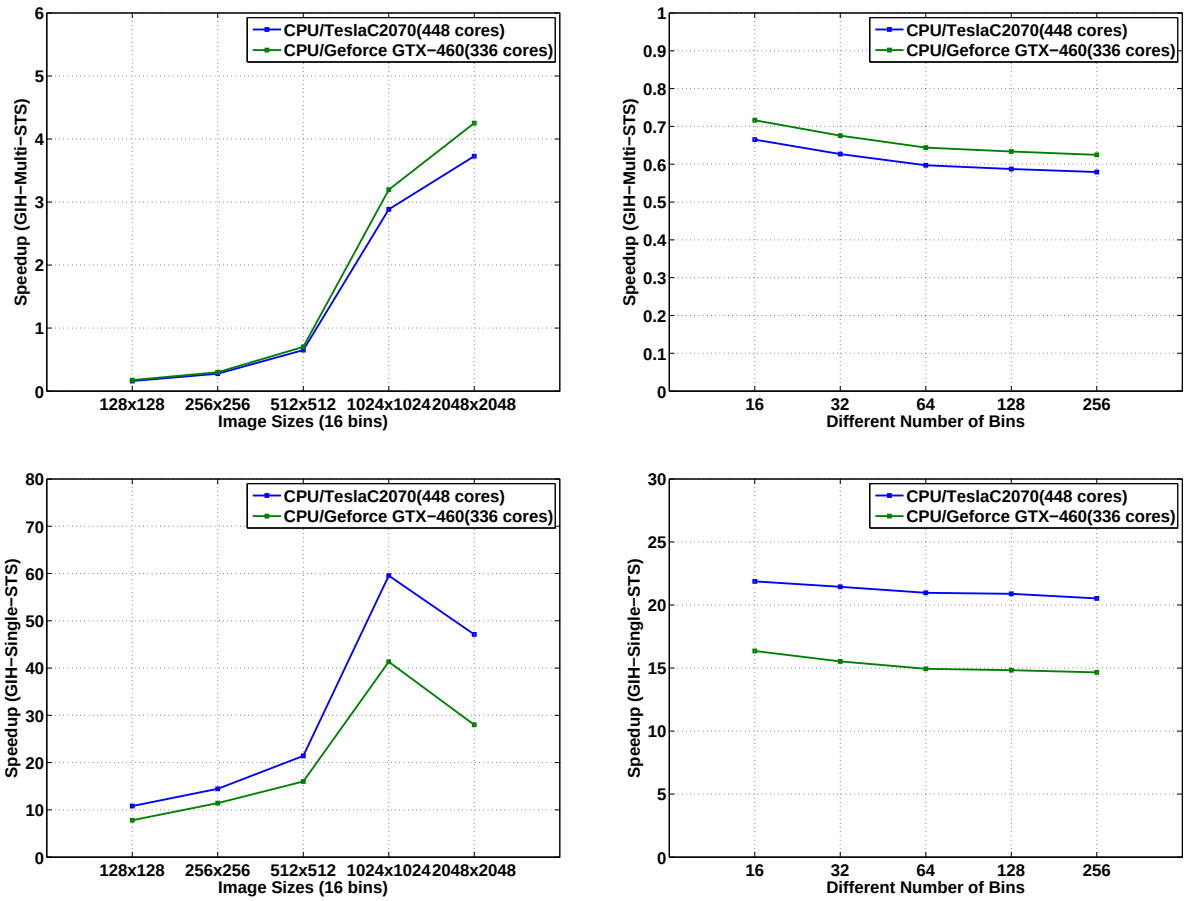



Figure 22: Speedup of the two GPU designs over CPU on two NVIDA graphic cards: (UL) Speedup of GIH-Multi-STS (with respect to CPU-only) with different image sizes, (UR) Speedup of GIH-Multi-STS with varying number of bins, (LL) Speedup of GIH-Single-STS for different image sizes, (LR) Speedup of GIH-Single-STS with varying number of bins for 512x512 image size.

implementation which was slower than the sequential version and the first implementation of GIH-Single-STS, clearly demonstrates that in parallelizing sequential image analysis algorithms on the GPU, data structures, GPU utilization and communication patterns need careful consideration. The GIH-Single-STS (*efficient communication*) implementation reduced the severe communication overhead bottleneck, by transferring an image size 1-D array instead of an integral histogram 3-D array and increased the GPU utilization. The GIH-Single-STS exploits an efficient prescan and 3-D transpose operation with maximum occupancy per kernel. It achieved frame rate of 185 for standard images $640 \times 480$ for 16 bins integral histogram computations which outperforms results presented for 8 SPEs (120 fr/sec for cross-weave scan and 172 for wavefront scan mode) in.[47] However, in most cases our performance is data-transfer-bound. We achieved a speedup factor of 21 times for $512 \times 512$ images (194 fr/sec) and 60 times for a $1K \times 1K$ image (49 fr/sec) for the integral histogram computation. One approach to further improve the time and memory efficiency of the GPU-based integral histogram method is to develop a custom parallel scan kernel for the horizontal and vertical cumulative sum computations without transpose phase for each tile of integral histogram tensor. We are continuing to evaluate two additional kernel mappings of the integral histogram computation GPU to use different data organization and scheduling methods that critically affects utilization of GPU resources (cores and shared memory). Tiling the 3-D array into smaller regular data blocks significantly speeds up the efficiency of the computation compared to a strip-based organization.[48] Preliminary results indicate that the tiled integral histogram using a diagonal wavefront scan has the best performance of about 104 frames/sec for $640 \times 480$ images and 32 bins with a speedup factor of about 41 compared to a single threaded sequential CPU implementation. Double-buffering is exploited to overlap computation and

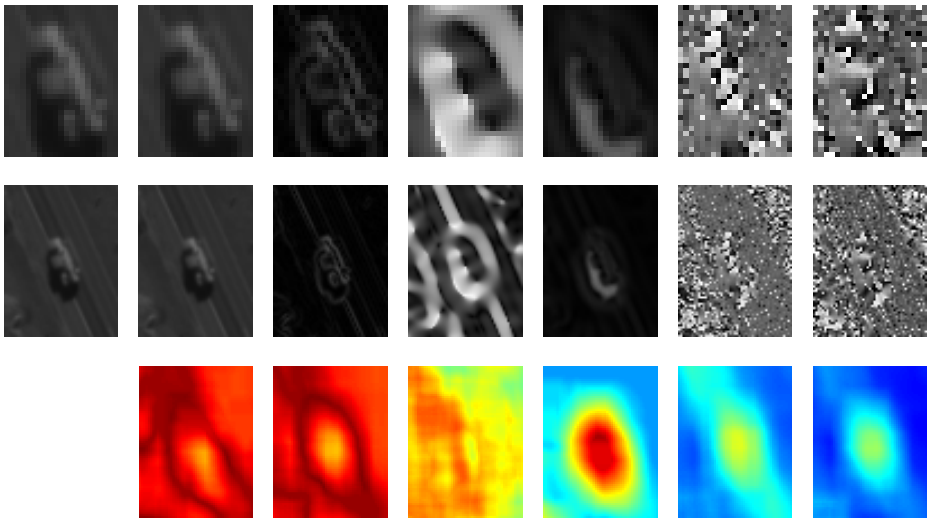


Figure 23: Top row shows the car template and associated raw target features for intensity, gradient magnitude, Hessian shape index, normalized curvature index, Hessian eigenvector orientations, and oriented gradient angles. Row 2 shows the predicted search window and associated raw features. Row 3 shows the corresponding likelihood maps combining target template with the associated search window features using integral histogram.

communication across sequence of images. In the case of large scale images, mapping integral histogram bin computations on multiple GPUs enables us to process on the order of 32 gigabytes integral histogram data per frame with a frame rate of 0.73 Hz and speedup factor of 153X over single threaded CPU implementation and the speedup of 45X over 16-threaded CPU implementation.

## 5. CONCLUSIONS

The University of Missouri (MU) has implemented, tested, validated and benchmarked a scalable parallel implementations of the integral histogram algorithm critical for computer vision tasks for fast multiscale subwindow-based object searching, motion analysis and content-based image retrieval applications. Several integral histogram kernels using Compute Unified Device Architecture (CUDA) optimizations for many core Graphics Processing Units (GPU) were investigated. The integral histogram algorithm was also parallelized using the StarSs programming model in collaboration the Barcelona Supercomputing Center for several architectures including Cell/B.E., GPU and SMP. These algorithms and code implementations have been delivered to AFRL for transition to other AFRL programs including CETE, E2AT and Next Generation WAMI. An initial implementation of the 3D spatiotemporal median filter for background model-based fast motion detection using integral histograms was also tested and showed promising performance improvements especially for medium to large images. The Cell/B.E. implementation of the integral histogram using cross-weave scan and 16 bins for a 640x480 image reaches 160 fr/sec using 8 SPEs. The wavefront scan for the same sized image reaches almost 200 fr/sec but critically depends on the block size. The GPU implementation of the integral histogram was 60 times faster than the sequential CPU version for a 1K x 1K image reaching 49 fr/sec and 21 times faster for 512 x 512 images reaching 194 fr/sec.

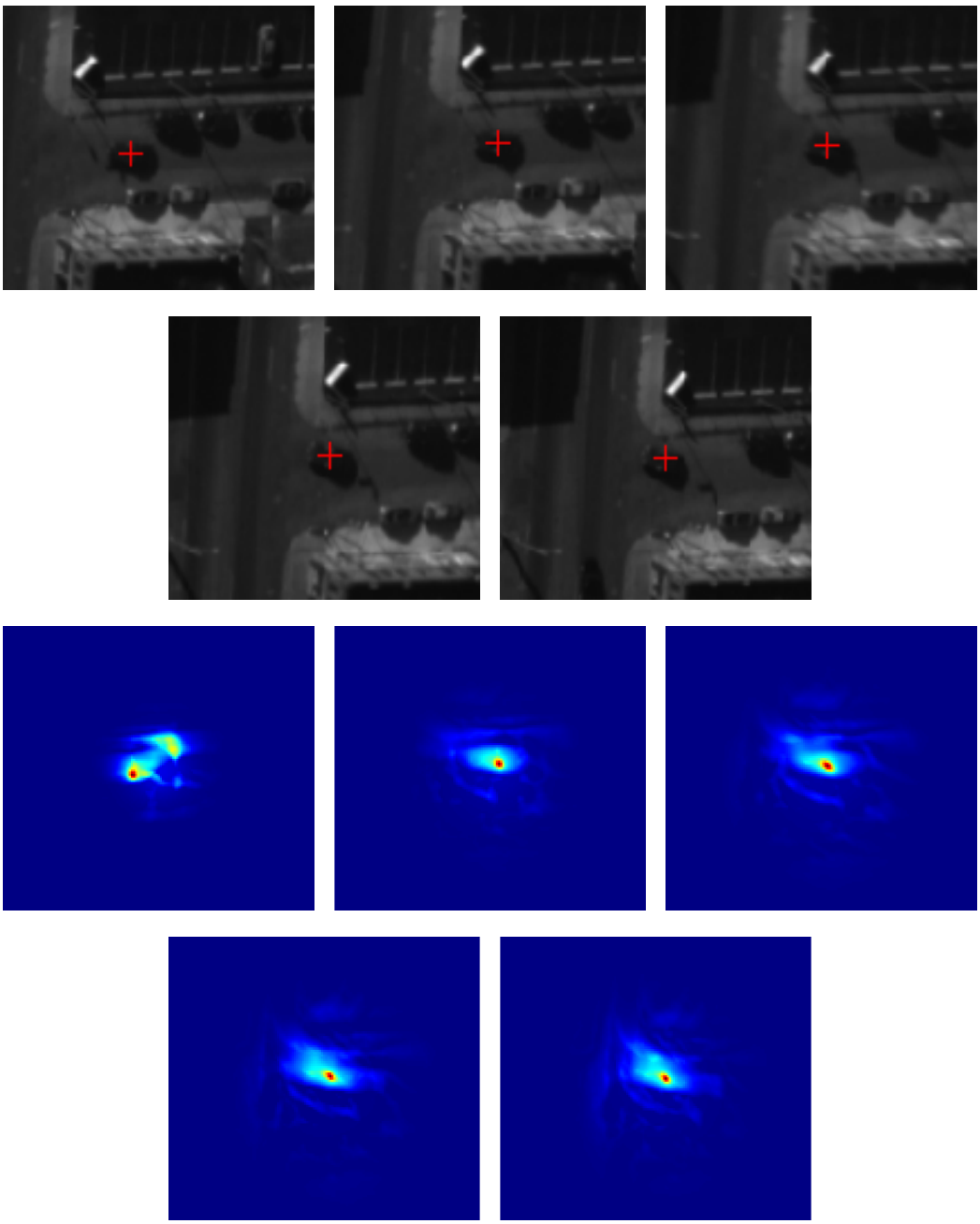


Figure 24: LOFT tracking results are shown for the first five frames for car C4_1_0 from CLIF aerial wide-area motion imagery.[2] Top row shows the tracked car locations and the bottom row shows the fused likelihood maps used by LOFT[3] to determine the best target location in each corresponding frame.

## 6. REFERENCES


[1] Harris, M., Sengupta, S., and Owens, J. D., "Parallel prefix sum (scan) with CUDA," *GPU Gems 3* **3**(39), 851–876 (2007).

[2] Air Force Research Laboratory, "Columbus Large Image Format (CLIF) dataset over Ohio State University," *https://www.sdms.afrl.af.mil/datasets/clif2007/* (2007).

[3] Pelapur, R., Candemir, S., Poostchi, M., Bunyak, F., Wang, R., Seetharaman, G., and Palaniappan, K., "Persistent target tracking using likelihood fusion in wide-area and full motion video sequences," in [*15th Int. Conf. Information Fusion*], (2012).

[4] Perez, J. M., Bellens, P., Badia, R. M., and Labarta, J., "Cellss: Making It Easier to Program the Cell Broadband Engine Processor," *IBM J. Res. Dev.* **51**(5), 593–604 (2007).

[5] Augonnet, C., Thibault, S., Namyst, R., and Nijhuis, M., "Exploiting the cell/be architecture with the starpu unified runtime system," in [*Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*], *SAMOS '09*, 329–339, Springer-Verlag, Berlin, Heidelberg (2009).

[6] Poostchi, M., Palaniappan, K., Bunyak, F., Becchi, M., and Seetharaman, G., "Realtime motion detection based on the spatio-temporal median filter using gpu integral histograms," in [*8th Indian Conference on Computer Vision, Graphics and Image Processing*], (2012).

[7] Poostchi, M., Palaniappan, K., Bunyak, F., Becchi, M., and Seetharaman, G., "Efficient gpu implementation of the integral histogram," in [*Lecture Notes in Computer Science (ACCV Workshop on Developer-Centered Computer Vision)*], **7728**(Part I), 266–278 (2012).

[8] Bellens, P., Palaniappan, K., Badia, R. M., Seetharaman, G., and Labarta, J., "Parallel implementation of the integral histogram," *Lecture Notes in Computer Science (ACIVS)* **6915**, 586–598 (2011).

[9] Bellens, P., Palaniappan, K., Badia, R. M., Seetharaman, G., and Labarta, J., "An integral histogram for parallel architectures," *IEEE Trans. Parallel and Distributed Systems* , Submitted (2013).

[10] Frigo, M., Halpern, P., Leiserson, C. E., and Lewin-Berlin, S., "Reducers and other cilk++ hyperobjects," in [*SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*], 79–90, ACM, New York, NY, USA (2009).

[11] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y., "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.* **30**(8), 207–216 (1995).

[12] Jenista, J. C., Eom, Y. H., and Demsky, B., "Ooojava: An out-of-order approach to parallel programming," in [*HotPar '10: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*], (2010).

[13] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable parallel programming with cuda," *Queue* **6**, 40–53 (March 2008).

[14] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for gpus: Stream computing on graphics hardware," *ACM TRANSACTIONS ON GRAPHICS* **23**, 777–786 (2004).

[15] Chu, S.-L. and Hsiao, C.-C., "Opencl: Make ubiquitous supercomputing possible," in [*Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*], *HPCC '10*, 556–561, IEEE Computer Society, Washington, DC, USA (2010).

[16] Pérez, J. M., Badia, R. M., and Labarta, J., "A dependency-aware task-based programming environment for multi-core architectures," in [*Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*], 142–151, IEEE (2008).

[17] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S., "Numerical linear algebra on emerging architectures: The plasma and magma projects," *Journal of Physics: Conference Series* **180**(1), 012037 (2009).

[18] Song, F., Yarkhan, A., and Dongarra, J., "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," tech. rep. (2009).

[19] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J., "Dague: A generic distributed dag engine for high performance computing," tech. rep., Innovative Computing Laboratory Technical Report, ICL-UT-10-01 (April 11, 2010).

[20] Song, F., Tomov, S., and Dongarra, J., "Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems," in [*Proceedings of the 26th ACM international conference on Supercomputing*], *ICS '12*, 365–376, ACM, New York, NY, USA (2012).

[21] Cosnard, M., Jeannot, E., and Yang, T., "Compact dag representation and its symbolic scheduling," *J. Parallel Distrib. Comput.* **64**(8), 921–935 (2004).

[22] Hennessy, J. L., Patterson, D. A., and Goldberg, D., [*Computer Architecture: A Quantitative Approach*], Morgan Kaufmann (2002).

[23] Tomasulo, R. M., "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Dev.* **11**, 25–33 (January 1967).

[24] Porikli, F., "Integral Histogram: A Fast Way To Extract Histograms In Cartesian Spaces," in [*in Proc. IEEE Conf. on Computer Vision and Pattern Recognition*], 829–836 (2005).

[25] Aldavert, D., de Mantaras, R. L., Ramisa, A., and Toledo, R., "Fast And Robust Object Segmentation With The Integral Linear Classifier," *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on* **0**, 1046–1053 (2010).

[26] Wei, Y. and Tao, L., "Efficient Histogram-Based Sliding Window," in [*CVPR*], 3003–3010 (2010).

[27] Blake, G., Dreslinski, R., and Mudge, T., "A survey of multicore processors," *IEEE Signal Processing Magazine* **26**, 26–37 (November 2009).

[28] Lin, D., Huang, X., Nguyen, Q., Blackburn, J., Rodrigues, C., Huang, T., Do, M., Patel, S., and Hwu, W.-M., "The parallelization of video processing," *IEEE Signal Processing Magazine* **26**, 103–112 (November 2009).

[29] Shams, R., Sadeghi, P., Kennedy, R., and Hartley, R., "A survey of medical image registration on multicore and the GPU," *IEEE Signal Processing Magazine* **27**, 50–60 (march 2010).

[30] Palaniappan, K., Bunyak, F., Kumar, P., Ersoy, I., Jaeger, S., Ganguli, K., Haridas, A., Fraser, J., Rao, R., and Seetharaman, G., "Efficient feature extraction and likelihood fusion for vehicle tracking in low frame rate airborne video," in [*13th Int. Conf. Information Fusion*], (2010).

[31] Mehta, S., Misra, A., Singhal, A., Kumar, P., Mittal, A., and Palaniappan, K., "Parallel implementation of video surveillance algorithms on GPU architectures using CUDA," in [*17th IEEE Int. Conf. Advanced Computing and Communications (ADCOM)*], (2009).

[32] Kumar, P., Palaniappan, K., Mittal, A., and Seetharaman, G., "Parallel blob extraction using the multi-core Cell processor," *Lecture Notes in Computer Science (ACIVS)* **5807**, 320–332 (2009).

[33] Grauer-Gray, S., Kambhamettu, C., and Palaniappan, K., "GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction," in [*5th IAPR Workshop on Pattern Recognition in Remote Sensing (ICPR)*], 1–4 (2008).

[34] Zhou, L., Kambhamettu, C., Goldgof, D., Palaniappan, K., and Hasler, A. F., "Tracking non-rigid motion and structure from 2D satellite cloud images without correspondences," *IEEE Trans. Pattern Analysis and Machine Intelligence* **23**, 1330–1336 (Nov. 2001).

[35] Perez, J. M., Badia, R. M., and Labarta, J., "Handling task dependencies under strided and aliased references," in [*Proceedings of the 24th ACM International Conference on Supercomputing*], *ICS '10*, 263–274, ACM, New York, NY, USA (2010).

[36] [*25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*], IEEE (2011).

[37] Bellens, P., Pérez, J. M., Badia, R. M., and Labarta, J., "Making the best of temporal locality: Just-in-time renaming and lazy write-back on the cell/b.e," *IJHPCA* **25**(2), 137–147 (2011).

[38] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J., "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters* **21**(2), 173–193 (2011).

[39] Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J., "Hierarchical task-based programming with starss," *Int. J. High Perform. Comput. Appl.* **23**(3), 284–299 (2009).

[40] Duran, A., Ayguade, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J., "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters (to be published)* (2011).

[41] Park, N., Hong, B., and Prasanna, V. K., "Tiling, block data layout, and memory hierarchy performance," *IEEE Transactions on Parallel and Distributed Systems* **14**, 2003 (2003).

[42] Ruetsch, G. and Micikevicius, P., "Optimizing matrix transpose in CUDA," *Nvidia CUDA SDK Application Note* (2009).

[43] UPC, "Paraver - parallel program visualization and analysis tool," (2000).

[44] UPC, "Paraver reference manual, draft 3.1." http://www.bsc.es/paraver (2001).

[45] Chan, A. L., "A Description on the Second Dataset of the U.S. Army Research Laboratory Force Protection Surveillance System," Technical Report ARL-MR-0670, Army Research Laboratory, Adelphi, MD, 2007 (2007).

[46] Palaniappan, K., Ersoy, I., and Nath, S. K., "Moving Object Segmentation Using The Flux Tensor For Biological Video Microscopy," in [*Proceedings of the Multimedia 8th Pacific Rim Conference on Advances in Multimedia Information Processing*], *PCM'07*, 483–493, Springer-Verlag, Berlin, Heidelberg (2007).

[47] Bellens, P., Palaniappan, K., Badia, R. M., Seetharaman, G., and Labarta, J., "Parallel implementation of the integral histogram," *LNCS (ACIVS)* **6915**, 586–598 (2011).

[48] Poostchi, M., Palaniappan, K., Li, D., Becchi, M., Bunyak, F., and Seetharaman, G., "Gpu kernel optimization for accelerating integral histogram computations," in [*IEEE Trans. Parallel and Distributed Systems*], Submitted (2014).

## 7. LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

AFRL - Air Force Research Laboratory

API - Application Programming Interface

C4ISR - Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance

Cell/B.E. - IBM Cell Broadband Engine multicore processor

CPU - Central Processing Unit

CUDA - Compute Unified Device Architecture (nVidia) item DMA - Direct Memory Access

EIB - Element Interconnect Bus

FLOPS - floating point operations per second

GFLOPS - gigaflops (billion floating point operations per second)

GPU - Graphics Processing Unit

IH - Integral Histogram

JDF - Job Definition Format

KB - kilobyte (1024 bytes)

LOFT - Likelihood of features tracking

LS - Local Store

MB - megabyte

MFLOPS - megaflops (million floating point operations per second)

MPI - Message Passing Interface

OpenMP - Open Multi-Processing

PCI - Peripheral Component Interconnect

PPE - Power Processing Element

SIMD - Single Instruction Multiple Data

SM - Streaming Multiprocessors

SMP - Symmetric Multi-Processing

SPE - Synergistic Processing Element

SPU - Synergistic Processing Unit

TDG - Task Dependency Graph

WAMI - Wide Area Motion Imagery